Computer Communications 56 (2015) 60–73

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Mitigating congestion based DoS attacks with an enhanced AQM technique

Harkeerat Bedi [a,*], Sankardas Roy [b], Sajjan Shiva [a]

[a] Dept. of Computer Science, University of Memphis, Memphis, TN 38152, United States
[b] Dept. of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, United States

ARTICLE INFO

Article history:
Received 8 August 2013
Received in revised form 5 July 2014
Accepted 10 September 2014
Available online 19 September 2014

Keywords:
Congestion control
Active queue management
Denial of service
Flow protection

ABSTRACT

Denial of Service (DoS) attacks are currently one of the biggest risks any organization connected to the Internet can face. Hence, the congestion handling techniques at the edge router(s), such as Active Queue Management (AQM) schemes must take into account such attacks. Ideally, an AQM scheme should (a) ensure that each network flow gets its fair share of bandwidth, and (b) identify attack flows so that corrective actions (e.g. drop flooding traffic) can be explicitly taken against them to further mitigate the DoS attacks. This paper presents a proof-of-concept work on devising such an AQM scheme, which we name Deterministic Fair Sharing (DFS). Most of the existing AQM schemes do not achieve the above goals or have significant room for improvement. DFS uses the concept of weighted fair share (*wfs*) that allows it to dynamically self-adjust the router buffer usage based on the current level of congestion, while aiding in identifying malicious flows. By using multiple data structures (a comprehensive repository and a cache) for keeping state of legitimate and malicious flows, DFS is able to optimize its runtime performance (e.g. higher bandwidth flows being handled by the cache). We demonstrate the performance advantage of DFS via extensive simulation while comparing against other existing AQM techniques.

1. Introduction

As widely evidenced, Denial of Service (DoS: regular as well as distributed) is one of the most prominent attack mechanisms on the Internet. In a recent study conducted by Radware and Ponemon Institute that consisted of surveying 705 IT practitioners, it was observed that 65% of the represented organizations suffered from three DoS attacks on average in 2012 [1]. Their average downtime lasted 54 min, resulting in an estimated cost of $22 K per minute, including the loss in revenue, traffic and end user productivity [1].

As computer hardware becomes cheaper and social networking becomes more accessible through the cyberspace, organization and execution of such distributed attacks become significantly easier. For example, botnets capable of performing DoS attacks of throughput ranging from 10–100 Gbps can be rented on the Internet for $200 per 24 h [2]. Moreover, the bandwidth used by these attacks is constantly growing. In 2014, a DDoS attack approximating 400 Gbps was observed by CloudFlare [3].

In general, DoS attacks can be classified into two categories depending on the layer of the OSI model they target: infrastructure-based attacks and application-based attacks. The former targets the layers 3 and 4 (i.e. network and transport layers) and the latter targets the application layer. Infrastructure-based attacks include SYN floods, UDP floods, ICMP floods, and IGMP floods, while application-based attacks include HTTP/SSL GET and POST floods, and NTP floods. Quarterly reports by Prolexic for the last several years show that DDoS attacks on the network infrastructure (Layer 3 and 4) far surpassed those that occurred on the application layer [4–6]. During the first quarter of 2014, over 87% of DDoS attacks were focused on the network infrastructure with UDP floods being one of the most popular attacks [6].

Congestion based attacks still dominate the denial of service landscape. Arbor Networks report that 61% of DDoS attacks observed by their survey respondents in 2014 were congestion based (the remaining included state-exhaustion and application layer attacks) [7]. Academia and the Industry have performed considerable amount of work towards understanding and mitigating network congestion based DoS attacks [8,9]. Active Queue Management (AQM) techniques are one of the most prominent approaches used for this purpose. Major network equipment providers including Cisco [10], Juniper [11] and Huawei [12] offer built-in support for several of them.

In this work, we design a novel congestion identification and mitigation technique that works at the infrastructure level. It aims

* Corresponding author.
E-mail addresses: hsbedi@memphis.edu (H. Bedi), sroy@ksu.edu (S. Roy), sshiva@memphis.edu (S. Shiva).

http://dx.doi.org/10.1016/j.comcom.2014.09.002
0140-3664/

to (a) ensure that each network flow gets its fair share of bandwidth, and (b) identify attack flows so that corrective actions (e.g. drop flooding traffic) can be explicitly taken against them. In particular, we develop an AQM technique called Deterministic Fair Sharing (DFS) that maintains per-flow state such that DoS attack traffic can be precisely identified and effectively mitigated while ensuring fairness.

Internet traffic can be broadly categorized into two major categories. The first category consists of flows that are inherently responsive in nature. That is, when they observe congestion due to packet loss or receive Explicit Congestion Notification (ECN) [13] marked packets, they reduce their sending rate. Examples of these include flows that use TCP as their transport protocol. They are known as *responsive* flows. The second category consists of flows that do not respond to congestion notifications. That is, they do not change their sending rates when they observe a packet loss or an ECN marked packet. Examples of these include flows that use UDP as their transport protocol. They are known as *unresponsive* flows.

1.1. Attacks

DoS attack flows can be defined as: (i) the set of unresponsive flows that use more than their fair share of the bandwidth; (ii) the set of responsive flows that do respond to congestion notifications, but not in a fair manner. Examples include flows using a hacked version of the TCP protocol with the intent to make the flows behave selfishly (or unresponsively); and (iii) the set of responsive flows that originate from a single client where their cumulative share is more than the fair share per client. In this case, the attacker generates a number of parallel fair TCP connections to the target server with the intent to use a major portion of the bandwidth as a whole. Examples include download accelerator tools that create multiple parallel connections to a target server requesting for the same file (different parts) in order to increase the client download speed.

1.2. Main idea

Prior research at large focused on using heuristic and probabilistic measures for creating AQM techniques. Choosing such measures give the benefit of low operational overhead, at the expense of fairness. DFS uses a deterministic approach to address router-based congestion and therefore is able to provide higher fairness. By using multiple data structures, DFS is able to reduce its operational overhead.

DFS keeps track of incoming traffic on a per-flow basis. By performing such granular analysis, it is able to provide a higher degree of fairness to well-behaved flows and accurately identify and throttle misbehaving flows by either dropping their packets or marking them using ECN (see Fig. 1).

To reduce its operational expense, DFS uses two kinds of data structures to manage per-flow information. Most of the flows, including all responsive flows (e.g. TCP flows), are stored in a comprehensive repository. In this paper we use an in-memory B-tree data structure for this purpose. B-tree is chosen since it can have a large fan-out while bounding its height. This in turn helps in reducing the maximum time required in retrieving or updating a stored flow. Flows marked as unfair (e.g. high bandwidth UDP flows) tend to require a higher frequency of updates and are stored separately in an array of fixed size that acts as a cache, for faster access. It should be noted that the fairness, attack identification and mitigation capabilities provided by DFS are not tied with the data structures it uses. B-tree is just one of many data structures that can be used for storing flow states.

Motivation: This work explores the feasibility of the following objectives:

- The potential of handling per-flow processing in modern routers. Per-flow processing techniques can provide unique benefits such as guaranteed bandwidth for legitimate flows and zero miss-classification of such flows as malicious. It can also ensure accurate identification of DoS attack traffic so that corrective actions can be taken explicitly against them [14]. This is necessary since such attacks are primarily performed with malicious intent and they most often lead to tangible economic damage to their victims [1]. Moreover, newer DoS defense techniques should not only focus on attack mitigation but also on explicit identification of malicious flows (e.g. [15]).
- Use of efficient data structures and approaches for storing state or managing lookup tables [16–18].

Note that a shorter and preliminary version of this work has been published in [19].

2. Related work

AQM techniques can be broadly classified in two categories based on the type of traffic they can handle. The first category aims to provide fairness when the incoming traffic consists of only responsive flows (e.g. TCP flows). Typical techniques include RED [20], BLUE [21], and AVQ [22]. The second category aims to provide fairness when the incoming traffic consists of both responsive and unresponsive flows (e.g. TCP and UDP flows). Well known techniques include CHOKe [23], SFB [24], RED-PD [25], and FRED [26].

Dischinger et al. [27] studied the deployment of RED in residential broadband networks. Their study consisted of 1894 broadband hosts from 11 cable and DSL providers in North America and Europe. They observed that 26.2% of the DSL hosts demonstrated a RED-style drop policy on their upstream queues. Moreover, the three providers owned by AT&T (i.e., Ameritech, BellSouth, and PacBell) exhibited deployment rates ranging from 50.3% to 60.5%.

Random Early Detection (RED) estimates the level of congestion in the router's buffer and drops packets accordingly by maintaining an exponentially weighted moving average (EWMA) of the queue length. One of the limitations of RED is that it requires significant parameter tuning to obtain optimal results. Several techniques have been proposed to address this limitation. Adaptive RED (ARED) [28] and its revisions [29,30] extend the effectiveness of RED by allowing the RED parameters to be dynamically self-adjusted based on traffic load. BLUE [21] uses link utilization and packet loss information to handle buffer congestion instead of monitoring the queue length. Another limitation of RED includes its incapability to provide fairness against unresponsive flows. Various AQM techniques have been proposed that are based on (or function with) RED and aim to address this limitation. Examples include FRED, CHOKe, xCHOKe [31], RECHOKe [32], StoRED [33], RRED [34], and RED-PD. Zhang et al. propose CPR (Congestion Participation Rate) [15], a metric and a congestion control technique that can be deployed in routers to identify and mitigate low-rate distributed DoS (LDDoS) attacks. It requires RED as part of its operation.

Random Exponential Marking (REM) [35] measures congestion by a quantity defined as *price*, instead of performance measures like loss, delay or queue length. AVQ uses a virtual queue that simulates the router buffer. If the virtual queue overflows, then the incoming packet either ECN-marked or discarded. CHOKe is a stateless technique that also tries to handle unresponsive flows. For each incoming packet at a congested router, a random packet is chosen from the router queue and if they both belong to the

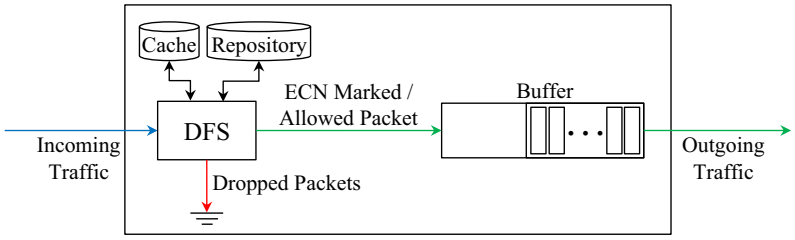

Fig. 1. A router running DFS, our proposed AQM technique. Per-flow state of high-bandwidth malicious flows are stored in a cache while state of others are stored in a comprehensive repository. Incoming traffic is observed for fairness before it is allowed through the buffer.

same flow, both are dropped. CHOKe modifies the packet headers during its operation. Schemes such as xCHOKe and RECHOKe have also been proposed to improve upon CHOKe. However, they all require additional storage space in the form of a lookup table to keep track of high bandwidth flows. Yamaguchi and Takahashi [36] propose PUNSI (Penalizing UNresponsive flows with Stateless Information) which aims to improve upon CHOKe. They identified that during the beginning of congestion, the packets from an unresponsive flow are accumulated more towards the tail of the buffer than its head. Since CHOKe performs its packet section by assuming a uniform distribution of packets in the buffer, it is unable to penalize unresponsive flows effectively.

Alvarez-Flores et al. [37] propose a congestion control mechanism called Drop-Sel to be applied to AQM schemes, which performs selective packet dropping based on the class of incoming traffic (real-time UDP flows (VoIP), non-real-time UDP flows and TCP flows). They use QoS and QoE for their performance assessment on VoIP flows. Chan et al. [38] propose an active buffer management scheme aimed towards improving the QoS of multimedia traffic by reducing congestion by detecting and dropping packets in multimedia streams with high jitter. They classify traffic into TCP and UDP and use RED for controlling TCP traffic and use jitter detection (JD their proposed solution) for controlling UDP traffic.

Stochastic Fair Blue (SFB), an extension of BLUE, uses bloom filters to maintain states for all flows by mapping them to a set of bins. Since more than one flow can be mapped to a set of bins, SFB is prone to misclassification of legitimate flows as being malicious. Stochastic RED (StoRED), based on RED, keeps track of individual flows by using a time-varying hash function to map flows to different counting bins. Since it uses hash functions for identification of individual flows, StoRED is also prone to misclassification of flows in congestion scenarios.

RED with Preferential Dropping (RED-PD) identifies high bandwidth flows by using the RED drop history. However, RED-PD suffers from the possibility of misclassification of responsive flows as being unresponsive, and may fail to identify unresponsive flows. Flow Random Early Drop (FRED) is a state-full technique that keeps track of flows whose packets are currently traversing the router buffer. When the buffer is small and the malicious flows cannot have at least one packet in the buffer at all given times, FRED is unable to identify them as malicious. Core-Stateless Fair Queuing (CSFQ) [39] functions by categorizing routers in a network into two types: one consisting of boundary routers and the other forming the core of a network as island routers (similar to an ISP network). The boundary routers add additional information in the headers of incoming packets that are then assessed by island routers to perform preferential dropping to enforce fairness. This approach requires packet header modification to include an extra field. Moreover, the routers present in an island are required to be modified to act on this extra field information. A similar framework is proposed in [40] which distributes different bookkeeping functions between access (edge) routers and core routers to provide per sender fairness and handle congestion based DDoS attacks.

Table 1 provides a comprehensive comparison of various techniques with DFS. The second column illustrates the amount of state these techniques require to store. The terms *si*, *spr*, *di*, *dpr*, and *pid* denote source IP, source port, destination IP, destination port, and protocol ID, respectively and they together represent the kind of flow information stored. Techniques marked as "‡" do not explicitly mention their flow classification information. Those marked as "∗" require a change in existing infrastructure or modify packet headers.

3. Deterministic Fair Sharing (DFS)

We begin our discussion of DFS by providing an overview of our approach. We then explain our algorithm in detail and provide a mathematical analysis. It should be noted that this AQM technique is not network-specific and is applicable whenever the following assumptions hold:

- The network router under consideration is able to process the packet headers of all incoming packets. It is able to keep per flow state for all active flows traversing through it. That is, the network bandwidth is the bottleneck of the system and not the packet processing speed. Most router manufacturers now offer per-flow state maintaining capabilities (e.g. NetFlow [41]), J-Flow [42] and NetStream [43]).
- The router has no knowledge of whether the flow is coming from the attacker or a legitimate user. Its belief on the legitimacy of the flow decreases with the increase of unfairness in its flow rate.
- We do not consider the case where an attacker might spoof the source address uniquely for each packet in a single flow. In the case that the spoofed source address is same for the entire flow the defense mechanism would act the same as if there were no spoofing. Spoofing can be avoided by using anti-spoofing methods such as [44], ingress filtering [45] or link-layer security protocols [46,47].
- We do not address a perfect DDoS attack that is created by using a large number of flows where each flow by itself is responsible and fair. In particular, we consider attacks as discussed in Section 1.1.

3.1. Overview

An ideal AQM technique is one that is able to identify each and every malicious flow when it behaves so. To ensure least damage, which includes denial of service for legitimate flows, it should drop all incoming traffic from the malicious flows, while protecting legitimate flows. It should not misclassify legitimate flows as malicious. Moreover, while allowing no incoming packets from malicious flows into the buffer, it should also be able to acknowledge their change in behavior if they become legitimate in the future. In case of no attack, an ideal AQM technique should be able to provide a high degree of fairness among competing legitimate flows while maintaining overall queue stability. Keeping per-flow state

Table 1
Comparison of existing congestion control techniques. The terms *si, spr, di, dpr, and pid* denote source IP, source port, destination IP, destination port, and protocol ID respectively.

Techniques	Stores flow state?	Robust to UDP-based DoS?	Guarantees fair share to legitimate flows?
RED	No	No	No
ARED	No	No	No
BLUE	No	No	No
AVQ	No	No	No
FRED	All flows in buffer (*si*, *spr*, *di*, *dpr*, *pid*)	Yes	No
CHOKe*	No	Yes	No
xCHOKe*,‡	High bandwidth flows	Yes	No
RECHOKe‡	High bandwidth flows	Yes	No
RED-PD	High bandwidth flows (*si*, *spr*, *di*, *dpr*, *pid*)	Yes	No
SFB	No	Yes	No
StoRED	No	Yes	No
NetFence*	Per-(sender, bottleneck link) (access routers)	Yes	Yes
CSFQ*	All flows (edge routers) (*si*, *di*)	Yes	Yes
RRED	No	No	No
CPR	All flows (*si*, *spr*, *di*, *dpr*, *pid*)	No	No
DFS	Per-flow (*si*, *di*)	Yes	Yes

is one such approach that makes it possible to achieve the aforementioned goals.

In this work we aim to build an AQM scheme that tries to meet the above ideal requirements while minimizing overhead. In case of no attack, the incoming traffic consists of only legitimate flows, that is, responsive (TCP) flows and unresponsive (UDP) flows within a fair rate.

TCP uses a feedback control algorithm known as additive-increase/multiplicative-decrease (AIMD) that makes these TCP flows highly sensitive to packet drops. Hence to ensure high degree of fairness among such competing flows, the packet drop (or ECN mark) decisions should be evaluated for each incoming packet, for each incoming flow. Dividing the available buffer space fairly among all incoming flows does not always guarantee an optimal solution. This is because if all flows were to use their fair buffer share, the buffer would always remain full. This would lead to higher latency and make the buffer unable to handle surges/fluctuations in incoming traffic. Moreover, it could lead to situations where packets from many flows will not be accepted even if the buffer has available empty space.

Thus an AQM technique is required to share the buffer fairly among all competing flows, while ensuring that the queue length always remains stable. Moreover, the buffer should not be under-utilized or overflown. To obtain these properties, we propose the concept of *weighted fair share* (*wfs*).

wfs determines the fair buffer share for each competing flow by using a scaling factor that is based on the available empty buffer space. It is a dynamic value that is updated for each incoming packet and is used to decide whether this packet should be allowed or dropped. It is defined as:

$$wfs = \frac{b}{n} \cdot \left(\frac{100}{q_p} - 1 \right) \tag{1}$$

Here q_p is the instantaneous current queue length (q) in percentage. That is, at a given time if the queue is half filled, then q_p is 50. Router buffer capacity is represented by b and n denotes the total number of active flows. The first factor of *wfs* that is "b/n" represents the fair share of buffer that each flow is allowed to have. It aids in providing fairness among competing flows. The second factor "$(100/q_p) - 1$" is a scaling factor that assigns a dynamic weight to the fair share of buffer (b/n). It helps in stabilizing the queue and preventing it from being full at all times, thus reducing latency.

Since *wfs* is evaluated for each incoming packet and uses the instantaneous queue length, it is able to dynamically self-adjust to provide queue stability and fairness. For example, if the current queue length is low, the scaling factor assigns a higher weight to the fair share of buffer allowed to a single flow. Hence, DFS behaves leniently and encourages the flows to have higher bandwidth. As a consequence, the current queue length begins to increase and weight assigned by the scaling factor begins to reduce, that in turn reduces the queue length. This self-adjusting behavior ensures queue stability and fairness. In particular, *wfs* ensures that the mean of the instantaneous queue length is always below the overall buffer capacity. However, because *wfs* is computed for each incoming packet, a high variance in the instantaneous queue length is observed. To smoothen the queue length and to reduce its high variance, DFS uses the exponential weighted moving average (EWMA) of *wfs*.

To ensure that the above properties of fairness and stability are met even during an attack, DFS maintains a variable called p_m for each flow that indicates its behavior. Higher p_m is assigned to higher unfair behavior by a particular flow.

An ideal AQM technique is also required to observe the change in behavior of the identified malicious flows, so that it can unmark them if they become legitimate in the future. Techniques that only look at the buffer usage to identify malicious flows and provide fairness need to allow some traffic from these identified malicious flows through the buffer in order to keep track of their behavior. Thus these approaches can only reach to a certain degree of fairness. In case of congestion based DoS attacks, allowing such traffic (even if limited in nature) can still cause the attack to succeed if the cumulative traffic from malicious flows becomes high.

To address this situation, DFS keeps track of the bitrates of the identified malicious flows. This allows it to monitor their behavior while allowing no traffic from them into the buffer. Since DFS already maintains per-flow state, keeping track of bitrates for a subset of flows does not add any significant cost. The extra cost consists of one add operation per packet and one division operation over a period of time.

3.2. Algorithm

DFS performs a series of operations before an incoming packet enters the buffer as well as after it leaves. These are defined by two functions namely `Enque` and `Deque`.

For each incoming packet p, the following three attributes are extracted: a) fid_p, b) $time_p$ and c) $size_p$. Here fid_p represents the flow ID used to uniquely identify flows. It consists of the sender and receiver's IP addresses and port numbers. In this paper DFS uses the IP addresses of the source and destination for identification of flows. Port numbers are not used as part of the definition so that

multiple TCP flows belonging to the same source address can be accounted for together. This can help in preventing attacks where an attacker creates multiple parallel TCP connections from a single node to a target victim with the motive to use more than their fair share of the bottleneck bandwidth. However DFS readily applicable to scenarios where flows are identified using the sender and receiver's IP addresses and port numbers. Attributes $time_p$ and $size_p$ represent the packet timestamp and size respectively.

For each flow stored, DFS maintains a series of attributes that are defined in Table 2. The behavior of a particular flow is determined by its p_m value which denotes its degree of unfairness. A flow is observed as unfair if its space in buffer (sz) exceeds the weighted fair share (wfs) at a given instance. When this occurs the value of p_m is increased and p_{time} which denotes when p_m was last updated is changed to current time.

DFS uses a set of parameters to evaluate and adjust p_m. The variables δ_i and δ_d represent the values by which p_m can be incremented and decremented. Two thresholds min_{th} and max_{th} bound the minimum and maximum values p_m can attain. These parameters together model the sensitivity of DFS towards high bandwidth flows. Using higher values for δ_i makes DFS more aggressive towards high bandwidth flows, whereas using higher values of δ_d makes it more lenient. The variable *freeze_time* (similar to the one used by [21]) represents a time period to wait before p_m is updated again. This time gap allows for the effect of such changes to be reflected back to the sender before the p_m is updated again. Hence its value should be based on the RTTs of flows multiplexed together. If a flow is observed as unfair for longer durations, its p_m value eventually reaches the max_{th} and it is marked as malicious by setting their Boolean *mark*. To punish malicious flows, DFS drops all incoming packets from these flows. Doing so gives DFS the opportunity to waste no bandwidth on malicious flows.

DFS also provides malicious flows the ability to correct their behavior in future. Since DFS does not allow any packets from marked malicious flows in the queue, to determine when these marked flows become good again, DFS keeps track of their bitrates. Thus in future if these flows reduce their bitrates to fair rates, DFS can identify the same and unmark them. The Boolean $store_{br}$ determines the flows whose bitrates are to be stored. The attribute *br* holds the stored bitrate (if $store_{br}$ is set). If one is not interested in this feature, then keeping track of bitrates of malicious flows can be disabled and it would not affect the other operations of DFS.

Table 2
Notation table.

Flow (i) variables	
fid_i	Flow ID (sender IP)
sz_i	Space occupied by flow in buffer
p_{m_i}	Flow unfairness indicator
p_{time_i}	Timestamp when p_m was last updated
$mark_i$	Boolean for marking flow as malicious
$store_{br_i}$	Boolean to store flow bitrate
br_i	Stored flow bitrate
DFS parameters (with default value)	
max_{th}	Maximum p_m threshold (=1)
min_{th}	Minimum p_m threshold (=0)
δ_i	p_m increment value (=0.04)
δ_d	p_m decrement value (=0.04)
freeze_time	Time to wait before updating p_m (=10 ms)
tolerance_factor	A factor used to identify *trending* flows (=3)
β	Beta scaling factor for EWMA of *wfs* (=0.05)
prune_interval	Frequency of executing the Prune function (=1 h)
Miscellaneous functions	
Allow(p)	Allow packet p to enter the queue
Drop(p)	Drop packet p
ECN-Mark(p)	ECN mark packet p (if ECN support is enabled)
EWMA(*wfs*)	Return EWMA of *wfs* based on β
ResetFlow(i)	Reset flow i's: $sz_i, mark_i, br_i, store_{br_i}$ to 0; and p_{m_i} to min_{th}
InsertNewFlow(p)	Insert a new flow i in B-tree with fid_i and sz_i matching the incoming packet p
Prune(*last_prune*)	Delete flow entries from data structures that have not been updated since time *last_prune*

Algorithm 1. Enque operation.

```
1: procedure ENQUE(p)                ▷ p is incoming packet
2:   if q ⩾ b then
3:     Drop(p)
4:   if matching flow i found in Cache or B-tree then
5:     if ProcessPacketAndDecide(p, i) then
6:       Drop(p)
7:     else
8:       sz_i ← sz_i + size_p
9:       Allow(p)
10:      if p_{m_i} ⩽ min_th & flow i ∈ Cache then
11:        move flow i to B-tree
12:      else if p_{m_i} ⩾ max_th & flow i ∈ B-tree then
13:        move flow i to Cache
14:    else
15:      InsertNewFlow(p)
16: end procedure
```

The `Enque` function is illustrated in Algorithm 1 and uses symbols defined in Table 2. During this operation, the fid_p of each incoming packet is extracted and the B-tree and Cache are search in sequence for any potential matches. Since these two data structures are mutually exclusive in terms of flows stored, if a match is found in Cache, then the B-tree is not searched. If a match is found, then the corresponding flow is updated and it is determined whether the current packet should be allowed through the buffer or not. This action is performed by the `ProcessPacketAndDecide` function. If the packet is to be allowed, then the value of the matching flow i's size in buffer (sz_i) is incremented by the packet's size ($size_p$). Similarly, when the packet is dequeued from the buffer, the `Deque` function decrements sz_i by $size_p$.

Flows are shifted between the B-tree and Cache based on their behavior – which is defined by the value of their p_m. If a flow is found in B-tree and has $p_m = max_{th}$, it is moved to Cache since it is a high bandwidth flow. If a flow is found in Cache and has $p_m = min_{th}$, it is moved to B-tree since this particular flow has changed from being malicious to a legitimate one. If the fid_p of the incoming packet does not match with any flow in B-tree or Cache, a new flow entry with matching fid_p is made in the B-tree.

For deletion of inactive flows from the Cache and B-tree, the `Purge` operation is performed periodically as illustrated in Algorithm 2. Its execution frequency can be adjusted by the variable *prune_interval*. During each operation, `Purge` deletes all flows that were not updated since it was last run. Since a flow is only updated when it transmits a packet, this operation ensures that inactive flows are effectively deleted.

Algorithm 2. Purge operation.

```
1: if current_time − last_prune > prune_interval then
2:     Prune(last_prune)
3:     last_prune = current_time
```

Algorithm 3. ProcessPacketAndDecide operation.

```
1: procedure ProcessPacketAndDecide(p, i)
2:   bool drop ← false
3:   time_gap ← time_p − p_time_i
4:   if p_m_i ⩽ min_th + (tolerance_factor · δ_i) then
5:     if sz_i > EWMA(wfs) then
6:       ECN-Mark(p)
7:       if time_gap > freeze_time then
8:         store_br_i ← 1
9:         p_m_i ← p_m_i + δ_i
10:        p_time_i ← time_p
11:    else if sz_i = 0 then
12:      p_m_i ← p_m_i − δ_d
13:      p_time_i ← time_p
14:      if p_m_i ⩽ min_th then
15:        ResetFlow(i)
16:  else
17:    ECN-Mark(p)
18:    if mark_i then
19:      drop ← true
20:    if time_gap > freeze_time & store_br_i ≠ 0 then
21:      if br_i > fair_bitrate then
22:        delta_factor ← br_i/fair_bitrate
23:        p_m_i ← p_m_i + (delta_factor · δ_i)
24:      else
25:        delta_factor ← fair_bitrate/br_i
26:        p_m_i ← p_m_i − (delta_factor · δ_d)
27:      p_time_i ← time_p
28:    if p_m_i > max_th then
29:      mark_i ← 1
30:  return drop          ▷ drop = true means p to be dropped
31: end procedure
```

The `ProcessPacketAndDecide` function determines whether the current incoming packet should be allowed or dropped. This decision primarily depends on the current value of the identified flow's p_m, its size in buffer and whether it has already been marked as malicious. This function is illustrated in Algorithm 3 using symbols defined in Table 2.

If a flow's p_m has incremented monotonically multiple times (in particular *tolerance_factor* times) in a sequence, then it's packet is ECN marked to notify the sender, its p_m is incremented by δ_i and its bitrate is stored. If the flow's size in buffer is less than the EWMA of *wfs*, then their packet is allowed. If it is zero, then its p_m is reset. DFS ensures that p_m of a flow is only incremented if the time since it was last updated is longer than the *freeze_time*. This setup ensures that responsive flows (senders) get sufficient time to receive the congestion notifications and the opportunity to reduce their sending rates. Hence, fairness among responsive flows is ensured by this step.

If the p_m has incremented more than the *tolerance_factor* times in a sequence, it indicates that the flow is deviating from its responsive behavior. This is so because, DFS allows a minimum of *freeze_time* gaps between p_m increments. This indicates that the sender did not respond to congestion notifications for at least *tolerance_factor* × *freeze_time* duration. We call these flows as *trending* flows since they have not yet been identified as either malicious or responsive. DFS keeps track of these flows by storing their bitrates. By storing their bitrates DFS get the opportunity to precisely adjust their p_m based on their degree of deviation from the fair rate. That is, instead of incrementing their p_m by δ_i, it is incremented by *delta_factor* times δ_i. The value of this *delta_factor* is directly proportional to difference between the flow's bitrate and the fair rate. By providing such control, DFS is able to increase the p_m of high bandwidth *trending* flows quicker. This in turn allows in faster identification of malicious flows. Higher value of *tolerance_factor* make DFS more lenient towards the *trending* flows, thus reducing the number of flow bitrates to keep track of. However, it also increases the time required to identify malicious flows.

If the *trending* or identified malicious flows begin to reduce their sending rates beyond the fair rate, then their p_m is decremented accordingly. When their p_m reaches min_{th}, DFS stops storing their bitrates and the formerly identified malicious flows are unmarked.

4. Analysis

We aim to analytically evaluate two kinds of complexities: (a) computation complexity, where we explain the operational costs and, (b) space complexity, where we show the amount of space DFS requires. We also evaluate correctness, where we show that DFS achieves its intended results.

4.1. Computational complexity

4.1.1. Primitive operations

Let the maximum number of flows traversing the router at any given point of time be $n = n_B + n_C$, where n_B and n_C are the total number of flows stored in B-tree and Cache respectively. This is constrained by the router memory size. During normal run, we assume that there are no malicious flows, and hence all flows are in B-tree and no flows are in the Cache.

The search, insert or delete in B-tree takes time of $O(t \cdot log_t n_B)$ whereas $O(t)$ is the intra-node search time, when t is the order of the tree. Note that t acts as a parameter – higher value of t implies lower number of levels in the tree, i.e., fewer nodes are required to traverse to search a key (i.e. $O(log_t n_B)$ node traversals). On the other hand, this also implies more keys (which is of $O(t)$) are stored per node that leads to higher intra-node search time.

During attack scenarios both B-tree and Cache tend to get populated by flows. The Cache being a fixed size array has a time complexity for each of search, insert and delete of $O(n_C)$. We envision that in most scenarios n_C would be of the same order of size as a B-tree node, i.e. $O(t)$.

A fixed size array is used as a cache, so that the search time, and space can be bounded and optimized. Having a shorter array size implies that malicious flows stored in it can be accessed faster, when compared to B-tree. Flows that use more than their fair share and are unresponsive to congestion notifications are identified as malicious and moved to the Cache. Since these flows send more packets when compared to legitimate flows, using the Cache for keeping their state reduces the cost of DFS.

4.1.2. Overall operational complexity

First, we highlight the costs when there is no ongoing attack.

Incoming Packet Processing: We begin by searching the Cache for a flow match. If the flow is not found, then the B-tree is searched instead. If the flow is not found in either of them, then a new flow is inserted. Therefore, each incoming packet can take time of the order $O(n_C)$ to $O(n_C) + O(t \cdot log_t n_B)$ depending on where the flow is stored. That means, in the worst case, the complexity of one packet processing is as follows:

$$O(n_C) + O(t \cdot log_t n_B) \tag{2}$$

Inserting a new flow in B-tree requires $O(t \cdot log_t n_B)$ time. However, since B-tree is searched after Cache; assuming n_C is of $O(t)$, the total cost is time of $O(t \cdot log_t n_B)$.

The above complexity holds when the Cache size is of the same order of size as a B-tree node. If a different order is set, following is the overall operational complexity. Let us consider that the Cache can store $(t \cdot \alpha)$ number of flows where a B-tree node can store t number of flows. Thus, when α is less than the height of the B-tree, the overall operational complexity of each incoming packet remains the same (as discussed above). Otherwise, the total cost becomes $O(t \cdot \alpha)$.

There is a tradeoff that is determined by α. If α is too low, then the potential Cache size becomes small and the likelihood of it being able to store all malicious flows becomes less. We assume that the number of malicious flows are limited in nature and will fit in Cache. On the other hand, if α is too high, the potential Cache size becomes large and the likelihood of underutilizing the Cache space becomes high.

Outgoing Packet Processing: For each outgoing packet the B-tree is searched first. This is done because DFS does not allow any packets to enter the queue from flows that are marked malicious. Since the malicious flows are stored in Cache and no packets from these flows enter the buffer, the outgoing packet consists of flows that are stored in the B-tree. Thus the time complexity for each outgoing packet is $O(t \cdot log_t n_B)$. The operations defined under "Miscellaneous functions" in Table 2 (other than `InsertNewFlow`) take time of constant order.

Inactive Flow Record Deletion Cost: Flows that have not received any packets since the last *prune_interval* period are deleted from either the B-tree or the Cache. This step ensures that the data structures only contain active flows and prevents the B-tree from growing indefinitely. Each `Prune` operation consists of traversing the data structures and deleting the inactive flows. B-tree has a traversal complexity of $O(n_B)$ and delete complexity of $O(t \cdot log_t n_B)$. Therefore the worst case complexity of pruning B-tree is $O(n_B \cdot t \cdot log_t n_B)$. Since the Cache is fixed in size, its worst case complexity is $O(n_C)$. It should be noted that the `Prune` operation is run once every *prune_interval* and not for each incoming packet.

Additional Operational Cost under Attack: Once a flow is identified as malicious, it is moved from B-tree to the Cache for faster access. A move operation requires 1 delete in B-tree and 1 insert operation in Cache, thus taking a total time of $O(n_C)$ + $O(t \cdot log_t n_B)$. Once shifted, the total lookup time of traffic from the attacking flows is reduced from $O(t \cdot log_t n_B)$ to $O(n_C)$, thus improving performance. Moreover, since DFS does not allow any traffic from such flows into the buffer, there is no outgoing packet processing overhead for such flows.

4.2. Space complexity

The number of flows stored per node in B-tree is of the order t. We envision the Cache such that it can store flows of the same order (t). This ensures that the Cache's space complexity is similar to a single B-tree node. If the space required for storing one flow is s_i bytes, then the Cache size would be $s_C = n_C \cdot s_i = t \cdot s_i$ bytes. B-tree uses space of $O(n_B)$. The byte size of B-tree is $s_B = n_B \cdot t \cdot s_i$. If the total memory (that houses the Cache and B-tree) available for storing flows is M bytes, then the height of the B-tree will be $h = log_t\left(\frac{M}{s_i} - t\right)$, when the order is t.

4.3. Correctness

DFS aims to provide two kinds of fairness. One being the fair and equal usage of the router buffer among all legitimate flows, and the other being fair and equal bottleneck link bandwidth usage. That is, if a router queue stabilizes at q_s KB, and the number of legitimate flows sharing the queue are n, then each flow gets $\frac{q_s}{n}$ KB of the queue on average over time. In steady state, this leads to each flow getting $\frac{R_p}{n}$ throughput, where R_p is the bottleneck link's total bandwidth capacity.

The following proof sketch shows that DFS is able to guarantee fair buffer share (with high likelihood as explained in Section 4.3.1) for each legitimate flow. DFS uses the concept of weighted fair share (*wfs*) to monitor and adjust the buffer usage of each flow. For each incoming packet, DFS compares the size of the flow in buffer corresponding to that packet with *wfs*. If the size of that flow is greater than *wfs* the packet is ECN marked. This allows DFS to ensure fairness when the traffic only consists of responsive TCP flows. Based on Eq. (1) (that represents *wfs*), it can be observed that as the percentage of queue length (q) increases, the buffer share per flow reduces. On the other hand, when q reduces, each flow is allowed more share of the buffer. This self-adjusting property of DFS ensures fairness among TCP flows and queue stability.

The effectiveness of DFS in self-adjusting the overall queue length of the router buffer can be demonstrated using the following example. If the queue is 20% full, then based on Eq. (1), we can observe that DFS behaves leniently and allows 4 times the fair buffer share for each flow. This encourages the flows to increase their sending rates, which eventually increases the overall queue length. When the queue length increases more than half the router buffer, DFS takes a stricter policy and dynamically reduces the fair share of buffer per flow. For example, if the queue is 80% full, then DFS only allows a quarter of the actual fair buffer share per flow, which makes the flows lower their sending rates, which eventually reduces the queue length. Thus, DFS is able to self-adjust the queue length by dynamically changing the allowed fair buffer share per flow, for each incoming packet.

4.3.1. Worst-case scenario for the buffer fair share

Let us quantify the impact resulting from the DFS policy. Referring to Eq. (1), strictly speaking, we cannot guarantee the per-flow share to be ideal, i.e., the $1/n^{th}$ part of the buffer where n is the total number of flows. Nevertheless, DFS exploits the multiplicity factor of the flows appearing simultaneously. Basically, DFS allows more room to a flow expecting that some other flows are not using their full share at the same point of time. Although in the worst case, we will not be able to give fair share to some unlucky flows, below we show that probability for the worst case is very low. We follow a similar methodology as the seminal work in [48].

Fig. 2 shows the buffer occupancy of the ith flow when it is in congestion avoidance mode. In this figure it can be observed that as the sender increases its congestion window, the buffer occupancy of its flow (X_{b_i}) also begins to increase (from point A). This occurs until the X_{b_i} hits the *wfs* limit, after which a segment is dropped or ECN marked from that flow by DFS. This leads the sender to reduce its congestion window by half and wait until it gets the acknowledgments for the outstanding packets. This waiting time period (i.e. one RTT [48]) is shown by BC, during which the X_{b_i} declines, until the sender gets acknowledgment for the packets it is waiting for. Then, the sender begins to send packets again which then increase X_{b_i} similarly.

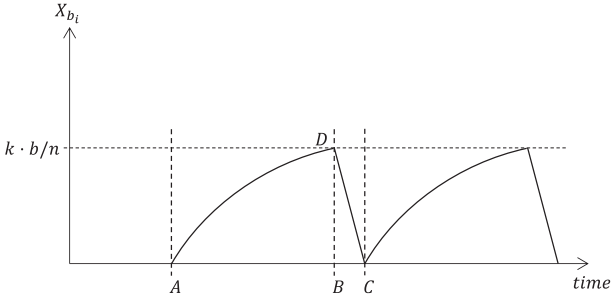

Fig. 2. Buffer occupancy of ith flow in congestion avoidance mode.

We assume that BC is very small when compared to the size of AB. The buffer occupancy of flow i at any point of time can be considered to follow an Uniform distribution as:

$$X_{b_i} \sim \cup\left[0, \frac{k \cdot b}{n}\right] \tag{3}$$

Here, k represents the *scaling factor* in Eq. (1) which can be also represented as:

$$k = \frac{b}{q} - 1 \tag{4}$$

We below derive the mean and variance of the dynamic queue length (q). For simplicity we assume that the traffic consists of only TCP flows that are in steady state (i.e. congestion avoidance mode).

The overall drop probability due to buffer overflow is:

$$P_r[q > b] = P_r\left[\sum X_{b_i} > b\right] \tag{5}$$

Let X_b represent the sum of all X_{b_i}'s (that means X_b is q).

$$X_b = X_{b_1} + X_{b_2} + \cdots + X_{b_n} \tag{6}$$

From the basic rules of probability, we get the following:

$$E(X_b) = E(X_{b_1}) + E(X_{b_2}) + \cdots + E(X_{b_n}) \tag{7}$$

$$= \left(\frac{k \cdot b}{2}\right) \tag{8}$$

From Eqs. (4) and (8), we get the dynamic queue size (q) at steady state as follows:

$$q = \frac{b}{2} \tag{9}$$

We can see that q is not expected to exceed b. With similar probability rules, we also get the variance of X_b, which is:

$$V(X_b) = \frac{1}{12}\left(\frac{b^2}{n}\right) \tag{10}$$

When n is sufficiently large, X_b's distribution (i.e. Irwin-Hall distribution) can be approximated as a normal distribution (by the Central Limit Theorem). Thus for a buffer of size 500 with 1000 flows, based on Eqs. (9) and (10), we can approximate mean as 250 and variance as 20.83 and get the probability $P(X_b > 500) \approx 0$. That means, the packet drop probability is negligible. Similar observations were made when the number of flows was increased to 100,000 and the buffer size was set to 1000 (i.e. $P(X_b > 1000) \approx 0$).

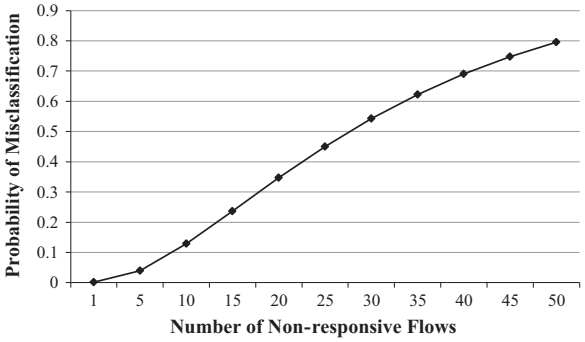

Fig. 3. Probability of misclassification by SFB.

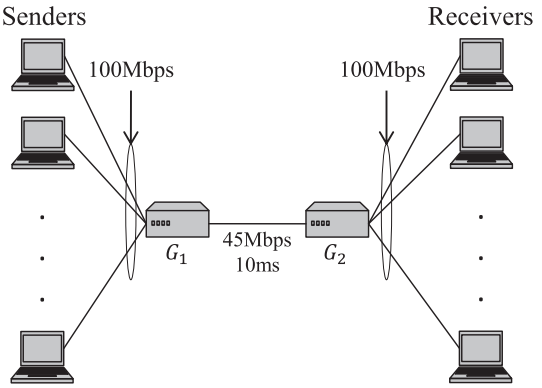

Fig. 4. Network topology in NS2.

Table 4
Link statistics for the experiment with single UDP Flow with CBR.

AQM	Queue stats.			Link stats.
	Avg. delay (ms)	S.D. delay (ms)	Avg. length (KB)	Utilization
DFS	10.9	2.7	61	1
SFB	12.3	4.2	69	0.9996
RED	9.7	4.6	55	1
FRED	35.4	1.3	199	1
SFQ	32.3	11.0	182	1
PI	35.5	0.3	199	1
BLUE	15.7	7.8	88	0.9989

AQM	Packet loss (Mbps)			Throughput (Kbps)			
	UDP	All TCP	Total	UDP	TCP avg.	TCP S.D.	Jain's Index (TCP)
DFS	45.01	0	45.01	0	113	16.2	0.98
SFB	44.84	0.28	45.12	160	112	18.8	0.97
RED	11.06	3.46	14.52	33,955	27	16.5	0.73
FRED	44.56	6.87	51.43	450	111	15.0	0.98
SFQ	43.12	7.17	50.29	1890	107	31.4	0.92
PI	7.81	2.41	10.22	37,191	19	11.6	0.74
BLUE	18.01	0	18.01	27,006	45	15.3	0.90

4.4. Comparison with Stochastic Fair BLUE (SFB) [24]

SFB uses a series of accounting bins to keep track of incoming flows. These bins are organized in L levels, with each level consisting of B bins. It uses L hash functions, each corresponding to one level of the accounting bins. For each incoming packet, the flow ID is mapped using each hash function and the corresponding bins are updated. Each bins maintains a unique variable p_m whose value depends on a function of its occupancy. SBF proposes that flows which use a higher (unfair) share of bandwidth, tend to pollute their corresponding bins. To provide fairness, SFB rate-limits all flows that map to those bins.

Since the number of bins is limited and constant, misclassification of flows can take place when the number of malicious flows

Table 3
AQM parameter settings.

DFS	$max_{th} = 1$, $min_{th} = 0$, $\delta_i = 0.04$, $\delta_d = 0.04$, $freeze\_time = 10ms$, $\beta = 0.05$, $tolerance_{factor} = 3$
SFB	$bins = 23$, $levels = 2$, $increment = 0.005$, $decrement = 0.001$, $hold\_time = 100$ ms, $pbox\_time = 50$ ms, $hinterval = 150$ s
SFQ	$maxqueue = 200$, $buckets = 23$
PI	$a = 0.00001822$, $b = 0.00001816$, $w = 170$, $q_{ref} = 100$

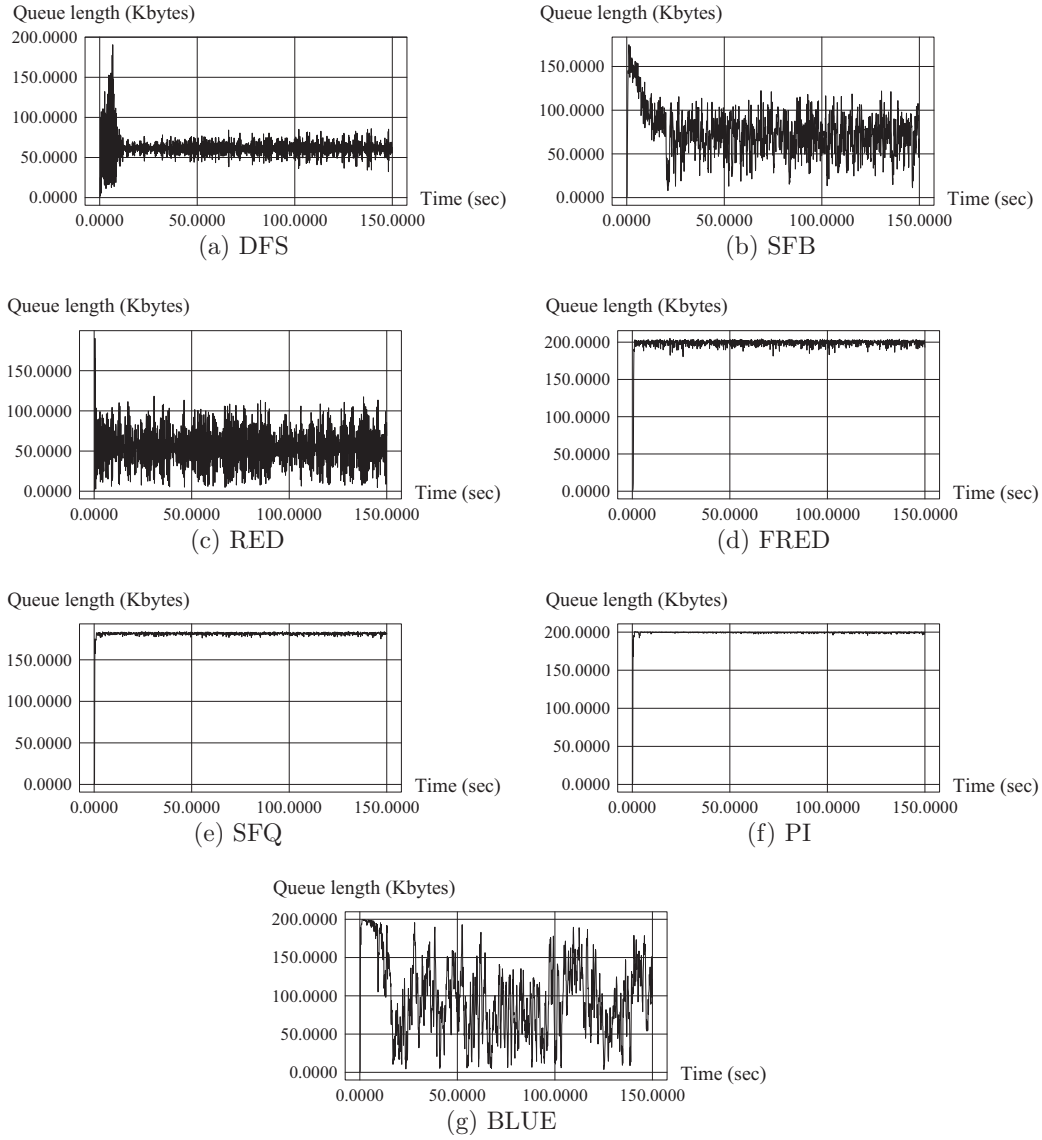

Fig. 5. Instantaneous queue usage during the experiment with single UDP flow with CBR.

Table 5
Variable bit rates for UDP flows.

Time (s)	Traffic rate (Mbps)
0–50	2
50–110	0.08
110–150	2

is large. Authors of SBF provide a closed-form equation in [24] that gives the probability that a well-behaved TCP flow gets misclassified as malicious as a function of L, B and M (number of malicious flows).

Fig. 3 shows the probability of misclassification using their closed-form equation for the SFB setup used in our evaluations (Section 5). In particular we used the parameter settings shown in Table 3 (i.e. $B = 23$ and $L = 2$). The number of malicious flows (M) were set to 25. In this scenario, it is evident that, as M exceed B, a significant portion of well-behaved flows can be misclassified as malicious (i.e. a false positive of 41%).

In our experimentation with SFB (Section 5.2) we observed that none of the well-behaved TCP flows were misclassified as malicious. However, 6 of the 25 well-behaved UDP flows were

Table 6
Link statistics for the experiment with multiple UDP flows with VBR.

AQM	Queue stats.			Link stats.
	Avg. delay (ms)	S.D. delay (ms)	Avg. length (KB)	Utilization
DFS	19.4	4.3	109	0.9989
SFB	12.3	14.8	49	0.7121
RED	9.7	4.5	54	0.9971
FRED	35.6	2.0	200	0.9990
SFQ	32.0	8.6	180	0.9990
PI	35.1	2.5	197	0.9981
BLUE	15.7	10.5	83	0.9476

AQM	Throughput (Kbps)						
	UDP avg.	UDP S.D.	G-UDP avg.	G-UDP S.D.	TCP avg.	TCP S.D.	Jain's Index (TCP)
DFS	41	0.8	99	0.6	104	13.9	0.98
SFB	936	417.0	77	34.7	17	12.9	0.63
RED	871	4.2	77	1.2	53	23.1	0.84
FRED	278	2.3	90	0.7	89	11.9	0.98
SFQ	691	181.0	86	6.4	63	39.1	0.72
PI	889	4.4	77	1.0	52	18.9	0.88
BLUE	705	5.2	64	1.1	59	15.9	0.93

misclassified as malicious (false positive = 24%). And only 6 of 25 malicious flows were accurately classified as malicious (false negative = 76%). In comparison, DFS misclassified none of the well-behaved TCP and UDP flows as malicious and accurately identified all the 25 malicious flows. That is, DFS exhibited no false positives or negatives.

To explain the above results: Note that SFB keeps no per-flow state and instead uses a series of hash functions that operate similar to a bloom filter to categorize flows within a fixed set of bins. This provides it the capability to look-up for each incoming packet in constant time and space. Since DFS keeps track of per-flow state, its operational overhead is a function of the number of active flows. In this case, the performance lost in operational overhead is gained by higher accuracy in malicious flow identification and lower well-behaved flow misclassification. However, DFS uses a combination of data structures to minimize operational overhead. It's complexity is analyzed in Section 4.1.

5. Evaluation

We conducted simulations to compare DFS with existing techniques. For this purpose, we implemented DFS and its underlying data structures (B-tree and Cache) with bookkeeping operations in NS2. We compare DFS with SFB [24], RED [20], FRED [26], SFQ [49], PI [50] and BLUE [21]. For those techniques in comparison, we used the code provided at the publicly available source [51] or their respective literature. Code for some AQM techniques [24,21] was taken from the AQM&DoS Simulation Platform that was created for the Robust Random Early Detection (RRED) algorithm [34].

We consider the network setup shown in Fig. 4 for our experimentation, which is similar to the one used by SFB in [24,21]. In particular, we use a dumbbell topology that consists of sender and receiver nodes connected to the gateway nodes G_1 and G_2, respectively, via duplex links. These two gateways are connected to each other by a duplex link, which is the bottleneck link in the network. The links between the senders and G_1 (and receivers and G_2) are 100 Mbps. The delay between the links is varied between a certain range such that different flows have different effective RTTs. The link between G_1 and G_2 has a capacity of 45 Mbps and a transmission delay of 10 ms. The bottleneck router is housed in G_1 with a buffer size of 200 KB. We use a packet size of 1 KB. The delay between the UDP sender nodes and G_1 and UDP receiver nodes and G_2 is 10 ms.

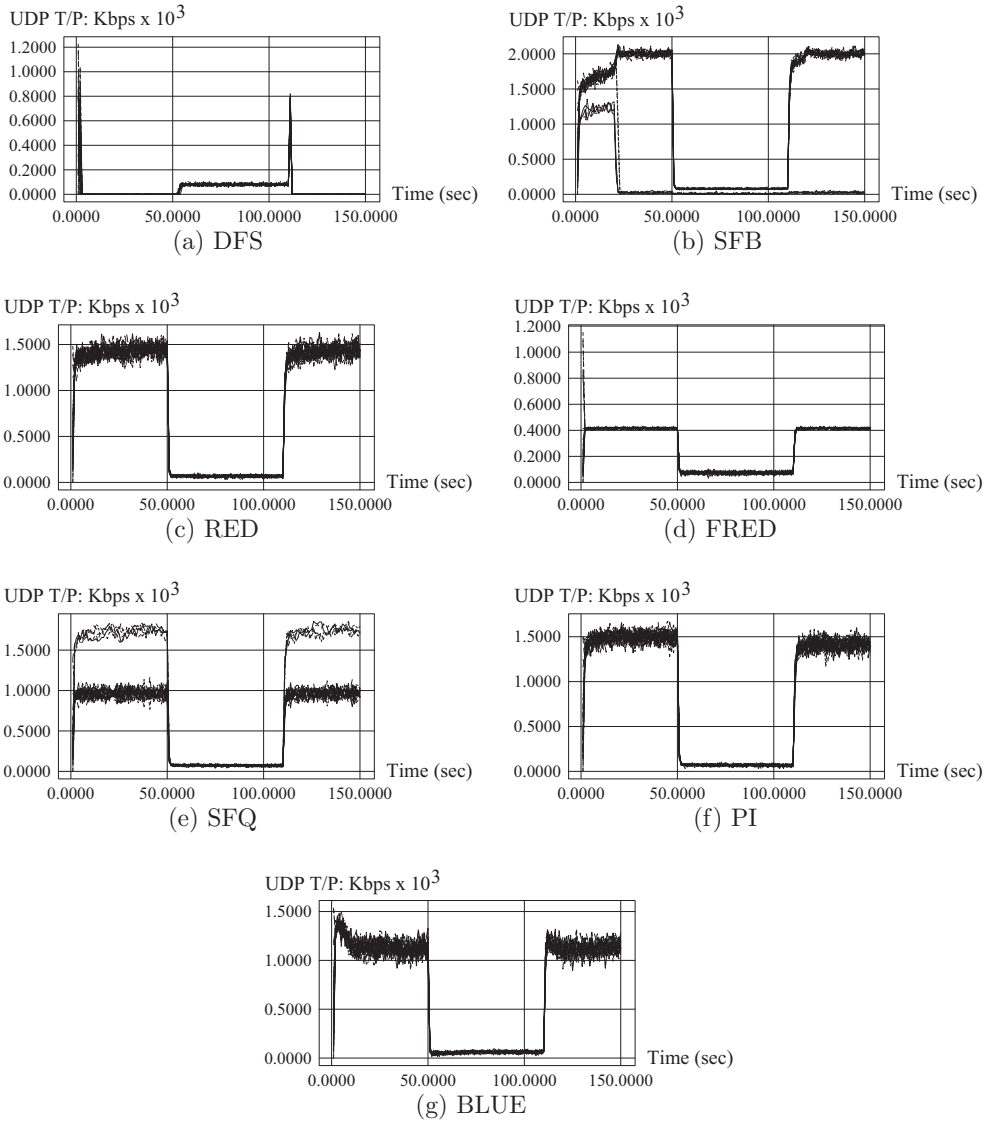

Fig. 6. Throughput of multiple UDP flows during the experiment with VBR.

Table 3 shows the parameter settings of the various AQM techniques used in our experimentation. The settings not shown in this table are set to the defaults given in their corresponding literatures. For our evaluation we conduct the following series of experiments where we vary the number of flows participating in an attack as well as their fairness behavior.

5.1. Single UDP Flow with Constant Bit Rate (CBR)

To perform a thorough comparison of DFS with others, we use the network topology and parameters used by [24,21] for evaluating their techniques. Doing so gives us the opportunity replicate their results as well as demonstrate the performance of DFS in their network setup. Our setup consists of one non-responsive (UDP) source with a CBR of 45 Mbps and 400 responsive (TCP) sources.

To make the setup closer to real-world scenarios, we extend this setup by using variable RTT for TCP flows. The delay between the TCP sender nodes and G_1 and TCP receiver nodes and G_2 is set to a random value between 1 and 15 ms. This results in an effective variable RTT for responsive flows between 24 and 60 ms. Support for ECN is also enabled across all AQM techniques we evaluate. We use TCP Selective Acknowledgment Options (TCP SACK) as our transmission protocol for responsive flows. The transmission protocol used by SFB in [24,21] was not mentioned in their publication. The experiment runs for 150 s and the first 50 s are ignored so that the collected statistics are based on the steady state traffic. Our findings are shown in Table 4.

We observe that the packet loss of the non-responsive UDP flow when using SFB, SFQ and RED is 44.84, 43.12 and 11.06 Mbps. These results are very close to those obtained by [21,24] during their evaluation (their observed packet loss was 44.84, 43.94 and 10.32 Mbps respectively). Similarly, we observe that the UDP flow throughput using SFB and RED is 160 and 33,955 Kbps, respectively, and is also very close to their results (i.e. 160 and 34,680 Kbps). However, we did observe a variation in the packet losses incurred by the TCP flows for these three techniques and the UDP throughput for SFQ from the results published in [21,24]. One of the reasons for this variation may include our choice of experiment parameter values that were not explicitly mentioned in their respective publications (e.g. TCP protocol used). In terms of throughput achieved by UDP flows, it is observed that DFS is able to identify the malicious UDP flow and throttle its throughput to 0 Kbps, which is the lowest among all. DFS is also able to provide the highest average throughput for TCP flows along with their Jain's Index value [52].

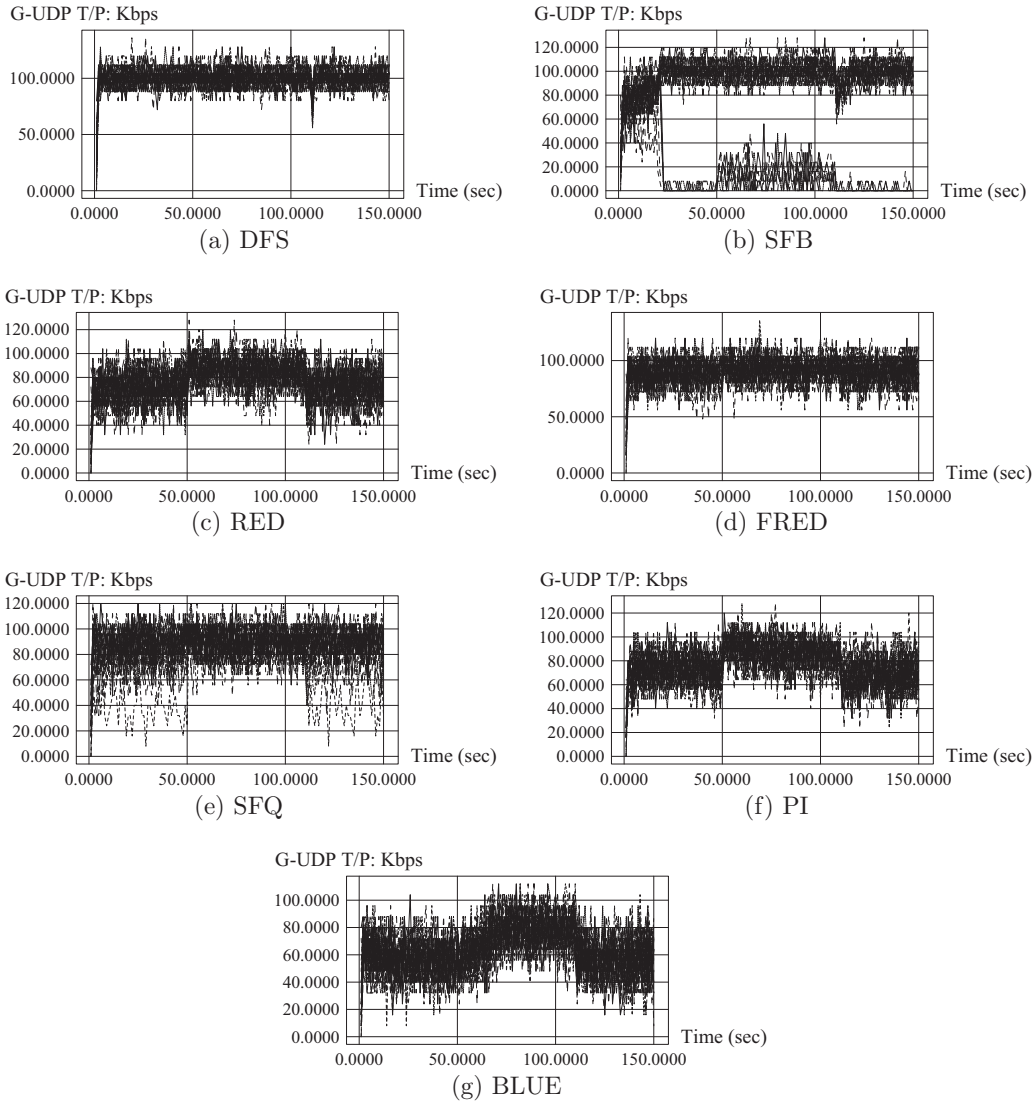

Fig. 7. Throughput of multiple G-UDP flows during the experiment with VBR.

Fig. 5 shows the instantaneous queue lengths of various AQM techniques during the entire 150 s of simulation and not just the steady state. We observe that DFS is able to stabilize its queue along with SFB and RED. Although RED provides a lower average queuing delay, it incurs a significant packet loss for TCP flows.

5.2. Multiple UDP flows with Variable Bit Rates (VBR)

In our second experiment we evaluate how these AQM techniques adapt to changes in throughput by malicious flows. As an example, we evaluate whether the flows are re-classified appropriately if they change their behavior from unfair to fair.

The experiment setup is similar to our previous experiment with the following changes. We have 25 UDP flows. These UDP flows fluctuate their traffic sending rate by being fair and unfair at certain times during the experiment. Table 5 shows the variable bitrates of these 25 UDP flows over time. That is, these UDP flows send data at the rate of 2 Mbps from 0–50 s and 110–150 s. During 50–110 s they send at the rate of 0.08 Mbps which is just below the fair rate (0.1 Mbps). We also have another set of UDP flows that send data at the fair rate of 0.1 Mbps throughout the experiment. We call these flows as good UDP flows (G-UDP). This experiment is conducted for 150 s. During our evaluation, the first 50 s of the experiment are not skipped so that the entire behavior of the flows can be observed.

Table 6 shows the link statistics of this experiment. We observe that DFS is able to provide the highest throughput to the TCP and G-UDP flows while stabilizing buffer usage and minimizing the throughput of the UDP flows.

The throughput obtained by the 25 UDP flows over the duration of the experiment is shown in Fig. 6. Fig. 6a shows the performance of DFS. We observe that soon after the experiment begins, all the UDP flows are accurately classified as malicious and their throughput is throttled to 0 Kbps. As they change their bitrate to 0.08 Mbps at the 50-s mark, they are re-classified as non-malicious and their traffic is allowed. When they change their sending rate back to 2 Mbps at 110-s mark, all of them are re-classified as malicious and throttled to 0 Kbps accurately as before.

During SFB, we observed that only a portion of UDP flows (6 out of 25) were identified as malicious. Moreover, SFB was unable to re-classify these UDP flows as non-malicious when their bitrates lowered below the fair share from 50 to 110 s. These flows can be identified in Fig. 6(b) as being throttled with their bitrates being sharply reduced at the 25-s mark and this continues till the end of

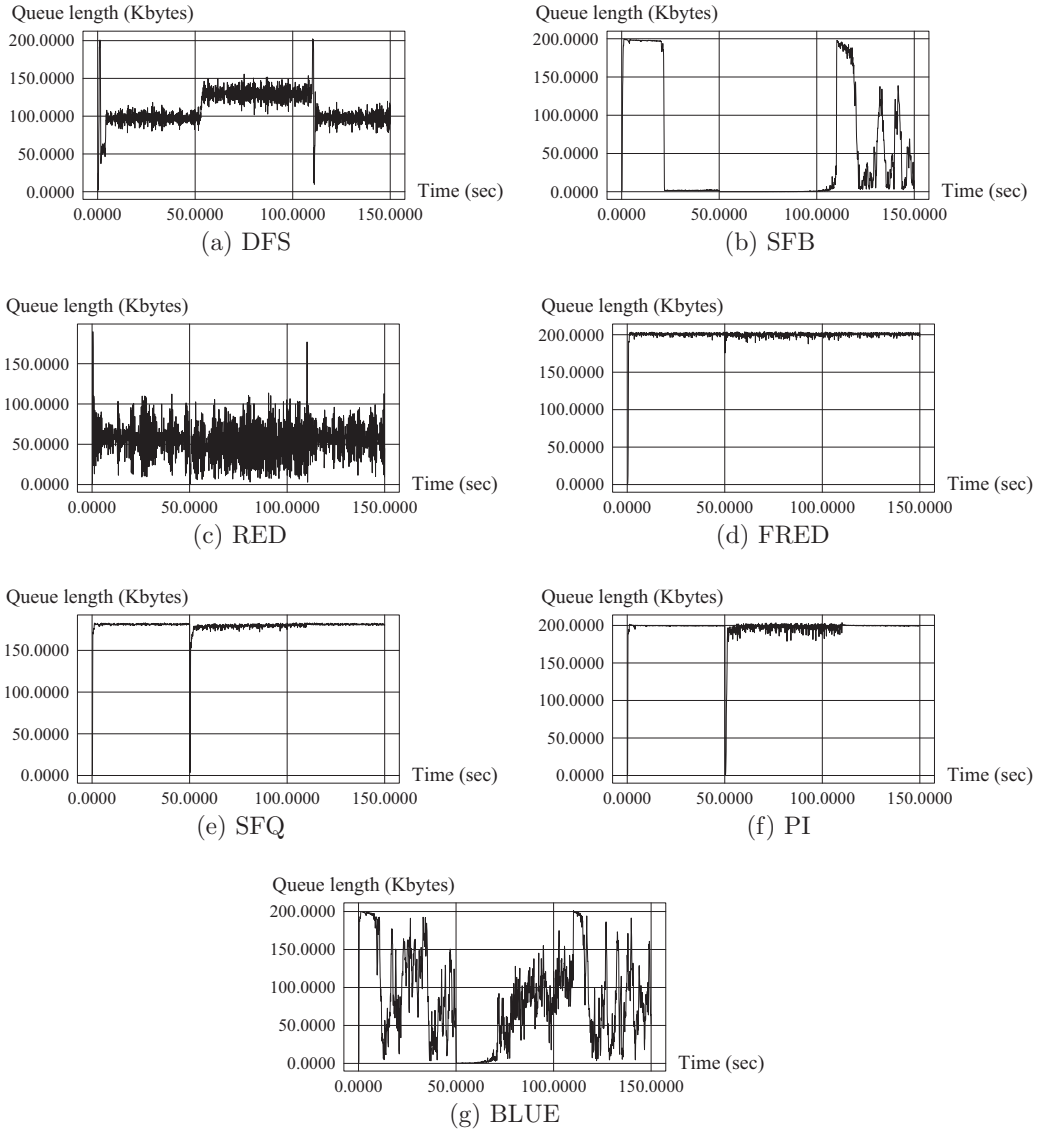

Fig. 8. Instantaneous queue usage during the experiment with multiple UDP flow with VBR.

the experiment. The remaining flows were not identified by SFB and thus remained unpunished.

Similarly Fig. 6(e) shows the UDP throughput when SFQ is used. We observed that a subset of UDP flows where throttled to approximately 1 Mbps and others remained above 1.5 Mbps. The remaining Fig. 6(c), (d), (f) and (g) shows the behavior of UDP flows when RED, FRED, PI and BLUE are used respectively.

Fig. 7 shows the throughput observed by the G-UDP flows during these AQM techniques. Fig. 7(b) shows the behavior of SFB. We observed that 6 out of 25 G-UDP flows were misclassified as malicious and their throughput was throttled down. These flows can be identified in Fig. 7(b) as those whose throughput sharply reduces at the 25-s mark. We can observe in Fig. 7(a) that DFS was able to provide a steady throughput to G-UDP flows despite the change in bitrates by the UDP flows. Other techniques shown in Fig. 7(c), (e), (f) and (g) (excluding FRED – Fig. 7d) demonstrated variations in throughput of G-UDP flow during the period UDP flows changed their throughput from being unfair to fair. Similarly Fig. 8 illustrates the instantaneous queue usage by these techniques.

6. Conclusion

In this work, we present a proof-of-concept towards devising a unique AQM technique that can successfully identify and mitigate congestion oriented DoS attack traffic. We propose a novel concept of *weighted fair share* that dynamically determines the fair buffer share for each competing flow to ensure optimal fairness. This concept is realized in DFS which uses a set of data structures in combination to provide low operational overhead while maintaining limited per-flow state and offer high DoS attack identification capability. We compare the performance of DFS with various other AQM techniques via extensive simulation in NS2. The results demonstrate that DFS is able to provide a higher degree of fairness and throughput to legitimate flows while stabilizing the router queue length and allowing the least bandwidth to the attack traffic. As part of an ongoing research, several extensions of DFS are being considered. We intend to test its performance by using different kinds of data structures for managing flow state. We also intend to evaluate its effectiveness using real world DoS attack traces.

References

[1] Radware, Ponemon Institute, Cyber Security on the Offense: A Study of IT Security Experts, 2012. <http://security.radware.com/uploadedFiles/ResourcesandContent/AttackTools/CyberSecurityontheOffense.pdf>.

[2] Verisign Inc., DDoS Mitigation – Best Practices for a Rapidly Changing Threat Landscape, 2012. <http://www.verisigninc.com/enUS/products-and-services/network-intelligence-availability/nia-information-center/ddos-best-practice-confirmation/index.xhtml>.

[3] M. Prince, Technical Details Behind a 400 Gbps NTP Amplification DDoS Attack, 2014. <http://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack>.

[4] Prolexic Quarterly Global DDoS Attack Report (Q4 2012), 2012. <http://www.prolexic.com/knowledge-center-ddos-attack-report-2012-q4.html>.

[5] Prolexic Quarterly Global DDoS Attack Report (Q4 2013), 2013. <http://www.prolexic.com/knowledge-center-ddos-attack-report-2013-q4.html>.

[6] Prolexic Quarterly Global DDoS Attack Report (Q1 2014), 2014. <http://www.prolexic.com/knowledge-center-ddos-attack-report-2014-q1.html>.

[7] Arbor Networks, 2014, Worldwide Infrastructure Security Report. <http://pages.arbornetworks.com/rs/arbor/images/WISR2014.pdf>.

[8] C. Douligeris, A. Mitrokotsa, DDoS attacks and defense mechanisms: classification and state-of-the-art, Comput. Networks 44 (2004) 643–666.

[9] T. Peng, C. Leckie, K. Ramamohanarao, Survey of network-based defense mechanisms countering the DoS and DDoS problems, ACM Comput. Surveys (CSUR) 39 (2007) 3.

[10] Cisco, Congestion Avoidance Overview, Online, 2013. <http://www.cisco.com/en/US/docs/ios/122/qos/configuration/guide/qcfconavps1835TSDProductsConfigurationGuideChapter.html>.

[11] Juniper Networks, Junos OS: Class of Service Configuration Guide, Online, 2013. <http://www.juniper.net/techpubs/enUS/junos12.2/information-products/topic-collections/config-guide-cos/config-guide-cos.pdf>.

[12] Huawei Technologies, Huawei AR150&200 Series Enterprise Routers: Configuration Guide – QoS, Online, 2012. <http://enterprise.huawei.com/ilink/enenterprise/download/HWU150660>.

[13] S. Floyd, TCP and explicit congestion notification, ACM SIGCOMM Comput. Commun. Rev. 24 (1994) 8–23.

[14] BBC News, Anonymous hacker group: Two jailed for cyber attacks, Online, 2013. <http://www.bbc.co.uk/news/uk-21187632>.

[15] C. Zhang, Z. Cai, W. Chen, X. Luo, J. Yin, Flow level detection and filtering of low-rate DDoS, Comput. Networks 56 (2012) 3417–3431.

[16] H. Lu, S. Sahni, A b-tree dynamic router-table design, IEEE Trans. Comput. 54 (2005) 813–824.

[17] K. Kim, S. Sahni, Efficient construction of pipelined multibit-trie router-tables, IEEE Trans. Comput. 56 (2007) 32–43.

[18] S. Hsieh, Y. Yang, A classified multisuffix trie for IP lookup and update, IEEE Trans. Comput. 61 (2012) 726–731.

[19] H. Bedi, S. Roy, S. Shiva, Mitigating congestion-based denial of service attacks with active queue management, in: IEEE Global Telecommunications Conference (GLOBECOM), 2013.

[20] S. Floyd, V. Jacobson, Random early detection gateways for congestion avoidance, IEEE/ACM Trans. Networking (ToN) 1 (1993) 397–413.

[21] W.C. Feng, K. Shin, D. Kandlur, D. Saha, The BLUE active queue management algorithms, IEEE/ACM Trans. Networking (ToN) 10 (2002) 513–528.

[22] S. Kunniyur, R. Srikant, An adaptive virtual queue (AVQ) algorithm for active queue management, IEEE/ACM Trans. Networking (ToN) 12 (2004) 286–299.

[23] R. Pan, B. Prabhakar, K. Psounis, CHOKe-a stateless active queue management scheme for approximating fair bandwidth allocation, in: INFOCOM, 19th Annual Joint Conference of the IEEE Computer and Communications Societies, vol. 2, 2000, pp. 942–951.

[24] W.C. Feng, D. Kandlur, D. Saha, K. Shin, Stochastic fair blue: a queue management algorithm for enforcing fairness, in: INFOCOM, 20th Annual Joint Conference of the IEEE Computer and Communications Societies., vol. 3, 2001, pp. 1520–1529. doi: 10.1109/INFCOM.2001.916648.

[25] R. Mahajan, S. Floyd, D. Wetherall, Controlling high-bandwidth flows at the congested router, in: 9th International Conference on Network Protocols (ICNP), IEEE, 2001, pp. 192–201.

[26] D. Lin, R. Morris, Dynamics of random early detection, in: ACM SIGCOMM Computer Communication Review, vol. 27, ACM, 1997, pp. 127–137.

[27] M. Dischinger, A. Haeberlen, K.P. Gummadi, S. Saroiu, Characterizing residential broadband networks, in: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, IMC '07, ACM, New York, NY, USA, 2007, pp. 43–56. <http://doi.acm.org/10.1145/1298306.1298313>, doi: 10.1145/1298306.1298313.

[28] S. Floyd, R. Gummadi, S. Shenker, et al., Adaptive RED: An Algorithm for Increasing the Robustness of RED's Active Queue Management, 2001, Preprint, <http://www.icir.org/floyd/papers.html>.

[29] W. Feng, D. Kandlur, D. Saha, K. Shin, Techniques for eliminating packet loss in congested TCP/IP networks, Ann Arbor 1001 (1997) 63130.

[30] W.C. Feng, D. Kandlur, D. Saha, K. Shin, A self-configuring RED gateway, in: INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings, IEEE, vol. 3, IEEE, 1999, pp. 1320–1328.

[31] P. Chhabra, S. Chuig, A. Goel, A. John, A. Kumar, H. Saran, R. Shorey, XCHOKe: malicious source control for congestion avoidance at Internet gateways, in: 10th IEEE International Conference on Network Protocols, 2002, Proceedings, 2002, pp. 186–187, doi: 10.1109/ICNP.2002.1181399.

[32] V. Govindaswamy, G. Zaruba, G. Balasekaran, RECHOKe: a scheme for detection, control and punishment of malicious flows in IP networks, in: IEEE Global Telecommunications Conference (GLOBECOM), 2007, pp. 16–21, doi: 10.1109/GLOCOM.2007.11.

[33] S. Chen, Z. Zhou, B. Bensaou, Stochastic red and its applications, in: IEEE International Conference on Communications, 2007, ICC'07, IEEE, 2007, pp. 6362–6367.

[34] C. Zhang, J. Yin, Z. Cai, W. Chen, RRED: robust RED algorithm to counter low-rate denial-of-service attacks, IEEE Commun. Lett. 14 (2010) 489–491.

[35] S. Athuraliya, S. Low, V. Li, Q. Yin, REM: active queue management, IEEE Network 15 (2001) 48–53.

[36] T. Yamaguchi, Y. Takahashi, A queue management algorithm for fair bandwidth allocation, Comput. Commun. 30 (2007) 2048–2059.

[37] E. Alvarez-Flores, J. Ramos-Munoz, P. Ameigeiras, J. Lopez-Soler, Selective packet dropping for VoIP and TCP flows, Telecommun. Syst. 46 (2011) 1–16.

[38] S.-P. Chan, C.-W. Kok, A. Wong, Multimedia streaming gateway with jitter detection, IEEE Trans. Multimedia 7 (2005) 585–592.

[39] I. Stoica, S. Shenker, H. Zhang, Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks, IEEE/ACM Trans. Networking (TON) 11 (2003) 33–46.

[40] X. Liu, X. Yang, Y. Xia, NetFence: preventing internet denial of service from inside out, ACM SIGCOMM Comput. Commun. Rev. 40 (2010) 255–266.

[41] Cisco, Introduction to Cisco IOS NetFlow, Online, 2012. <http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/prodwhitepaper0900aecd80406232.pdf>.

[42] Juniper Networks, 2011, Juniper Flow Monitoring. <http://www.juniper.net/us/en/local/pdf/app-notes/3500204-en.pdf>.

[43] Huawei Technologies, NetStream (Integrated) Technology White Paper, Online, 2012. <http://enterprise.huawei.com/ilink/enenterprise/download/HW201022>.

[44] T. Peng, C. Leckie, K. Ramamohanarao, Protection from distributed denial of service attacks using history-based IP filtering, in: IEEE International Conference on Communications (ICC), vol. 1, 2003, pp. 482–486.
[45] P. Ferguson, D. Senie, RFC 2827: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing, 2000.
[46] IEEE Standard 802.1X, 2001. <http://www.ieee802.org/1/pages/802.1x.html>.
[47] X. Liu, X. Yang, Y. Lu, To filter or to authorize: network-layer DoS defense against multimillion-node botnets, ACM SIGCOMM Computer Communication Review, vol. 38, ACM, 2008, pp. 195–206.
[48] G. Appenzeller, I. Keslassy, N. McKeown, Sizing router buffers, ACM SIGCOMM Computer Communication Review, vol. 34, ACM, 2004, pp. 281–292.
[49] P. McKenney, Stochastic fairness queueing, in: INFOCOM '90, Ninth Annual Joint Conference of the IEEE Computer and Communication Societies. The Multiple Facets of Integration, Proceedings, IEEE, vol. 2, 1990, pp. 733–740, doi: 10.1109/INFCOM.1990.91316.
[50] C. Hollot, V. Misra, D. Towsley, W.-B. Gong, On designing improved controllers for aqm routers supporting tcp flows, in: INFOCOM 2001, Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, Proceedings, IEEE, vol. 3, 2001, pp. 1726–1734, doi: 10.1109/INFCOM.2001.916670.
[51] S. Thulasidasan, Contributed code – nsnam, 2012. <http://nsnam.isi.edu/nsnam/index.php/ContributedCode>.
[52] R. Jain, A. Durresi, G. Babic, Throughput Fairness Index: An Explanation, Technical Report, Tech. rep., Department of CIS, The Ohio State University, 1999.