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 its edge router(s), such as active queue management (AQM) schemes must consider possibilities of 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 a significant room for improvement. DFS uses the concept of weighted fair share (wfs) which allows it to dynamically self-adjust the router buffer usage based on the current level of congestion, while assuring fairness among flows and aiding in identifying the malicious ones. We demonstrate the performance of DFS via extensive simulation and compare against other existing AQM techniques.

I. INTRODUCTION

As widely evident, Denial of Service (DoS: regular as well as distributed) is one of the most prominent attack mechanisms on the Internet. A recent study that consisted of surveying 705 IT practitioners observed that 65% of the organizations suffered from three DoS attacks on average during 2012. Their average downtime lasted 54 minutes, resulting in an estimated cost of $22K 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 Internet, organizing and launching such distributed attacks have become tremendously easier. Botnets capable of performing DoS attacks of 10-100 Gbps can be rented on the Internet for about $200 per 24 hours [2]. Moreover, the bandwidth used in these attacks is constantly growing. In 2012, attacks peaking at 100.8 Gbps were observed by Arbor Networks [3]. In 2013, this number has increased to 300 Gbps [4].

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 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. Prolexic [5] observed that over 81% of attacks encountered by their clients were targeted on infrastructure with UDP floods being one of the most popular.

In this work, we design a unique congestion identification and mitigation technique that works at the infrastructure level. It aims to (a) ensure that each network flow gets its fair share of the bandwidth, and (b) identify the attack flows so that the corrective actions (e.g. dropping their traffic) can be explicitly taken against them. 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.

Prior research at large has mostly focused on using heuristic and probabilistic measures in creating AQM techniques. Choosing such measures gives the benefit of low operational overhead, at the expense of fairness. DFS uses a deterministic approach and hence is able to provide higher fairness. DFS 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 Explicit Congestion Notification (ECN).

Figure 1: A router running DFS, our proposed AQM technique. Per-flow state is kept in Cache/Main-storage. Incoming traffic is observed for fairness before it is allowed through buffer.

To minimize operational overhead, 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 the main-storage. In this paper, we consider B-tree as an example data-structure realizing this main-storage. B-tree is considered since it provides a large fan-out while bounding its height. This reduces the maximum traversal time required in retrieving or updating a flow stored in the B-tree. Flows marked as unfair (e.g. high bandwidth UDP flows) require a higher frequency of updates and therefore are stored in a fixed size cache for faster access. The fairness, attack identification...
and mitigation capabilities provided by DFS are independent of 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 aids in exploring the feasibility of: (a) Handling per-flow state in routers for mitigating and identifying congestion based attacks. Per-flow processing techniques can provide unique benefits such as guaranteed bandwidth for legitimate flows and zero misclassification of such flows as malicious. It can also aid in accurate identification of DoS attack traffic so that corrective actions can be taken explicitly against them [6]. (b) Use of efficient data structures and approaches for storing state or managing lookup tables (some examples are in [7], [8]).

II. RELATED WORK

AQM techniques are broadly classified in two categories based on the kind of traffic they can handle. The first category aims at providing fairness when the incoming traffic consists of only responsive flows (e.g. TCP flows). Typical techniques include RED [9], BLUE [10], and AVQ [11]. The second category aims at providing fairness when the incoming traffic consists of both responsive and unresponsive flows (e.g. TCP and UDP flows). Well known techniques include CHOKe [12], SFB [13], RED-PD [14], and FRED [15].

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. BLUE uses link utilization and packet loss information to handle buffer congestion instead of focusing on the queue length. A limitation of RED includes its incapability to provide fairness against unresponsive flows. There are various AQM techniques that are based on (or function with) RED and aim to address this limitation. These include FRED, CHOKe, RECHOKe [16], RRED [17], and RED-PD. [18] propose CPR (Congestion Participation Rate), a metric and congestion control technique to identify and mitigate low-rate distributed DoS (LDDoS) attacks. It requires RED as part of its operation and maintains per-flow state.

CHOKe tries to handle unresponsive flows by keeping no per-flow state, with limited success. RECHOKe aims to improve upon CHOKe by using additional storage space to keep track of high bandwidth flows. gCHOKe [19] too aims to improve upon CHOKe but maintains no per-flow state.

Stochastic Fair Blue (SFB) is an extension of BLUE that uses bloom filters for maintaining state of all flows by mapping them to a set of bins. Since more than one flow can be mapped to a particular set of bins, SFB is prone to misclassification. DFS prevents the likelihood of misclassification by using efficient data structures for keeping limited per-flow state.

RED with Preferential Dropping (RED-PD) identifies high bandwidth flows by using RED drop history but with limited success. 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.

III. DETERMINISTIC FAIR SHARING (DFS)

The proposed AQM technique is not network-specific and is applicable with the following assumptions: (a) 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. Most router manufacturers now offer per-flow state maintaining capabilities (e.g. NetFlow [20]). (b) DFS has no knowledge if a flow is coming from an attacker or a legitimate user. Its belief on the legitimacy of a flow decreases with the increase of unfairness in its flow rate. (c) We do not consider the case where an attacker might spoof source addresses. Anti-spoofing methods such as [21], [22] can aid in such scenarios. (d) We do not address a distributed DoS (DDoS) attack that is created by using a large number of flows where each flow by itself is responsible and fair.

A. 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 DoS for legitimate flows; it should drop all incoming traffic from these identified 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 these flows 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 is one approach that makes it possible to achieve the above-mentioned goals. In this work we build an AQM scheme that tries to meet the above requirements by keeping limited per-flow state information while minimizing overhead.

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 underutilized 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:

\[ \text{wfs} = \frac{b}{n} \left(100 \cdot \frac{1}{q_p} - 1\right) \] (1)

Here \(q_p\) is the instantaneous current queue length (q) in percentage, \(b\) is the router buffer capacity 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. wfs ensures that the mean of the instantaneous queue length is always below the overall buffer capacity. Since 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 marking probability called \( p_m \) for each flow that indicates its behavior. Higher \( p_m \) denotes 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 maintains per-flow state, keeping track of bitrates for a subset of flows does not add a significant cost. The extra cost consists of one add operation per packet and one division operation over a period of time.

### B. 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 \( \text{Enque} \) and \( \text{Deque} \).

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, which is the sender and receiver IP address of the packet. Port numbers are not used as part of the definition so that multiple TCP flows belonging to the same source and destination addresses 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. 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 I. 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 \( \text{pt} \) which denotes when \( p_m \) was last updated is changed to current time.

DFS uses a series of constants to evaluate and adjust \( p_m \). The constants \( \delta \) and \( \delta_d \) represent the values by which \( p_m \) can be incremented and decremented. Two thresholds \( \text{max}_{th} \) and \( \text{min}_{th} \) bound the minimum and maximum values \( p_m \) can attain. These constants together model the sensitivity of DFS towards high bandwidth flows. Using higher values for \( \delta \) makes DFS more aggressive towards high bandwidth flows, whereas using higher values of \( \delta_d \) makes it more lenient.

The constant \( \text{freeze}_{time} \) (similar to the one used by [10]) 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 round-trip times of flows multiplexed together.

If a flow is observed as unfair for longer durations, its \( p_m \) value eventually reaches \( \text{max}_{th} \) and it is marked as malicious by setting their Boolean \( \text{mark} \). To punish malicious flows, DFS drops all incoming packets from them. This gives DFS the ability 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. We define fair rate as \( \text{fair}_{bitrate} = \frac{T}{n} \) where \( r \) is total incoming bitrate and \( n \) is total number of active flows. The Boolean \( \text{store} \) determines the flows whose bitrates are to be stored.

<table>
<thead>
<tr>
<th>Flow (s) Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>( fid_i )</td>
</tr>
<tr>
<td>( sz_i )</td>
</tr>
<tr>
<td>( pm_i )</td>
</tr>
<tr>
<td>( pt )</td>
</tr>
<tr>
<td>( mark_i )</td>
</tr>
<tr>
<td>( store_{br} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constants (\text{= default value})</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{max}_{th} )</td>
</tr>
<tr>
<td>( \text{min}_{th} )</td>
</tr>
<tr>
<td>( \delta_1 )</td>
</tr>
<tr>
<td>( \delta_d )</td>
</tr>
<tr>
<td>( \text{freeze}_{time} )</td>
</tr>
<tr>
<td>( \text{tolerance}_{factor} )</td>
</tr>
<tr>
<td>( \beta )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Allow}(p) )</td>
</tr>
<tr>
<td>( \text{Drop}(p) )</td>
</tr>
<tr>
<td>( \text{ECN-Mark}(p) )</td>
</tr>
<tr>
<td>( \text{EWMA}(s) )</td>
</tr>
<tr>
<td>( \text{ResetFlow}(i) )</td>
</tr>
<tr>
<td>( \text{InsertNewFlow}(p) )</td>
</tr>
</tbody>
</table>

Table I: Symbol Tables
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.

Algorithm 1: Enque Operation

1. **procedure** Enque\((p)\) \( \triangleright p \) is incoming packet
2. \( i f \ q \geq b \) then
3. \( \text{Drop}(p) \)
4. \( i f \) matching flow \( i \) found in Cache/B-tree then
5. \( \text{ProcessPacketAndFlow}(p, i) \)
6. \( \text{Drop}(p) \)
7. \( \text{else} \)
8. \( s_{z_i} \leftarrow s_{z_i} + \text{size}_p \)
9. \( \text{Allow}(p) \)
10. \( i f \ p_m \leq \text{min}_th \) & flow \( i \) in Cache then
11. move flow \( i \) to B-tree
12. \( \text{else if} \ p_m \geq \text{max}_th \) & flow \( i \) in B-tree then
13. move flow \( i \) to Cache
14. \( \text{end else} \)
15. \( \text{InsertNewFlow}(p) \)
16. **end procedure**

The Enque function is illustrated in Algorithm 1 and uses symbols defined in Table I. During this operation, the \( fid_p \) of each incoming packet is extracted and the B-tree and Cache are searched in sequence for any potential matches. Since these two data structures are mutually exclusive in terms of flows stored, if a match in 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 ProcessPacketAndFlow function.

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 \( p_m = \text{max}_th \), it is moved to Cache since it is a high bandwidth flow. If a flow is found in Cache and has \( p_m = \text{min}_th \), 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.

The ProcessPacketAndFlow 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 2 using symbols defined in Table I.

If the flow’s \( p_m \) has incremented less than the \( \text{tolerance}_\text{factor} \) times in a sequence, and its size in buffer is less than the EWMA of \( wfs \), then the packet is allowed. Else, the packet ECN marked to notify the sender, its \( p_m \) is incremented by \( \delta_i \) and its bitrate is stored. To prevent the \( p_m \) from being incremented too often, DFS ensures that \( p_m \) is only incremented if the time since it was last updated is longer than the freeze time. If the flow’s size in buffer is zero, then its \( p_m \) is reset. This setup ensures that responsive flows (senders) get sufficient time to receive the congestion notifications and the opportunity to reduce their sending rates. This step ensures fairness among responsive flows.

If the \( p_m \) has incremented more than the \( \text{tolerance}_\text{factor} \) 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 \( \text{tolerance}_\text{factor} \times \text{freeze time} \). 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. This gives DFS the ability to precisely adjust their \( p_m \) based on their degree of deviation from the fair rate (\( \text{fair}_\text{bitrate} \)). Therefore, instead of incrementing their \( p_m \) by \( \delta_i \), it is incremented by

### Algorithm 2: ProcessPacketAndFlow Operation

1. **procedure** ProcessPacketAndFlow\((p, i)\)
2. \( \text{bool} \ \text{drop} \leftarrow 0 \)
3. \( \text{time}_\text{gap} \leftarrow \text{time}_p - \text{p}_\text{time}_i \)
4. \( i f \ p_m \leq \text{min}_th + (\text{tolerance}_\text{factor}_i \times \delta_i) \) then
5. \( i f \ s_{z_i} > \text{EWMA}(wfs) \) then
6. \( \text{ECN-Mark}(p) \)
7. \( i f \ \text{time}_\text{gap} > \text{freeze}_\text{time} \) then
8. \( \text{store}_{br} \leftarrow 1 \)
9. \( \text{p}_m \leftarrow \text{p}_m + \delta_i \)
10. \( \text{p}_\text{time}_i \leftarrow \text{time}_p \)
11. \( \text{else if} \ s_{z_i} = 0 \) then
12. \( \text{p}_m \leftarrow \text{p}_m - \delta_d \)
13. \( \text{p}_\text{time}_i \leftarrow \text{time}_p \)
14. \( i f \ p_m \leq \text{min}_th \) then
15. \( \text{ResetFlow}(i) \)
16. **end else**
17. \( \text{ECN-Mark}(p) \)
18. **if** \( \text{mark}_i \) **then**
19. \( \text{drop} \leftarrow 1 \)
20. **if** \( \text{time}_\text{gap} > \text{freeze}_\text{time} \) & \( \text{store}_{br} \neq 0 \) **then**
21. \( i f \ \text{br}_i > \text{fair}_\text{bitrate} \) **then**
22. \( \text{delta}_\text{factor} \leftarrow \text{br}_i / \text{fair}_\text{bitrate} \)
23. \( \text{p}_m \leftarrow \text{p}_m + (\text{delta}_\text{factor} \times \delta_i) \)
24. **else**
25. \( \text{delta}_\text{factor} \leftarrow \text{fair}_\text{bitrate} / \text{br}_i \)
26. \( \text{p}_m \leftarrow \text{p}_m - (\text{delta}_\text{factor} \times \delta_d) \)
27. \( \text{p}_\text{time}_i \leftarrow \text{time}_p \)
28. **if** \( \text{p}_m > \text{max}_th \) **then**
29. \( \text{mark}_i \leftarrow 1 \)
30. **return** \( \text{drop} \) \( \triangleright \text{drop} = 1 \) means \( p \) to be dropped
31. **end procedure**
delta_factor times \( \delta_t \). The value of 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 \( p_n \) 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.

IV. Analysis

A. Computational Complexity

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.

Search, insert and delete in B-tree takes time of \( O(t \cdot \log n_B) \) 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, thus, fewer nodes are required to traverse to search a key (i.e. \( O(\log n_B) \) node traversals). This also implies more keys (which is of \( O(\delta) \)) are stored per node that leads to higher intra-node search time.

The Cache being a fixed size array has a time complexity for search, insert and delete of \( O(n_C) \). We envision that in most real life scenarios \( n_C \) would be of the same order of size as a B-tree node, i.e. \( O(t) \). However, we use \( n_C \) during our analysis to make them more general.

2) 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 \( O(n_C) + O(t \cdot \log n_B) \). Once shifted, the total lookup time of traffic from the attacking flows is reduced from \( O(t \cdot \log n_B) \) 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.

B. 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 \( h = \log_t \left( \frac{M}{s_i} \right), \) when the order is \( t \).

V. Evaluation

We conducted simulations to compare DFS with existing techniques like SFB and RED. For this purpose, we implemented DFS and its underlying data structures (B-tree and Cache) with bookkeeping operations in NS2. For other techniques, we used the code provided at the publicly available source [17], [23] or their respective literature. The parameters used for DFS are shown in Table I under the heading “Constants”. For SFB, the parameters used were: \( bins = 23, levels = 2, increment = 0.005, decrement = 0.001 \), \( hold_time = 100ms \), \( pbox_time = 50ms \) and \( hinterval = 150s \). Those not listed were set to their defaults.

We consider a network setup shown in Figure 3, which is similar to that used by SFB in [13], [10]. The delay between links is varied such that different flows have different effective round-trip times (RTT). The bottleneck router is housed in \( G_1 \) with a buffer size of 200 KB. We use a packet size of 1KB.

This experiment evaluates how the AQM techniques adapt to changes in throughput by malicious flows. We test how well (a) the buffer usage is stabilized, (b) legitimate flows are protected against congestion, (c) unfair flows are accurately classified as malicious, and (d) malicious flows are re-classified appropriately when they become fair again.

The experiment consists 400 fair TCP flows and 25 UDP flows. These UDP flows send data at the rate of 2 Mbps from 0-50 seconds and 110-150 seconds. During 50-110 seconds they send at the rate of 0.08 Mbps which is just below the fair rate (0.1 Mbps). The experiment also includes 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). The throughput obtained by these 25 UDP flows over the duration of the experiment is shown in Figure 2. Figure 2a shows the performance of DFS. We observe that soon after the experiment begins, all 25 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-second 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-second mark, they are all re-classified as malicious and throttled to 0 Kbps accurately.

In case of SFB, notice in Figure 2b the throughput of these 25 UDP flows is divided in two groups. This is because SFB was only able to identify some UDP flows (6 out of 25) as malicious instead of the expected all. This occurred since SFB uses Bloom filters to maintain flow state and hence is prone to misclassification. SFB was also unable to re-classify these UDP flows as non-malicious when their bitrates lowered below the fair share from 50-110 seconds. These flows can be identified in Figure 2b with their bitrates being sharply
reduced at the 25-second mark which continued till the end of the experiment. The remaining flows were not identified by SFB and thus were unpunished. Figure 2c shows the behavior of UDP flows when RED is used. Table II shows the link statistics (average and standard deviation) 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, identifying malicious flows and minimizing their throughput.

VI. CONCLUSION
This paper presents a proof of concept work towards devising a unique AQM technique that is aimed towards achieving a higher degree of fairness while mitigating potential congestion based DoS attacks. By using a series of data structures in combination, DFS is able to provide low operational overhead while maintaining limited per-flow state. We evaluate the performance of DFS via extensive simulation against various AQM techniques and demonstrate promising results.

REFERENCES