RECHOKe: A Scheme for Detection, Control and Punishment of Malicious Flows in IP Networks

Visvasuresh Victor Govindaswamy, Gergely Záruba and G. Balasekaran

Abstract—In this paper, we are proposing a scheme called RECHOKe (REpeatedly CHOose and Keep for malicious flows, REpeatedly CHOose and Kill for non-malicious flows) to be used for detecting, controlling and punishing of malicious flows in IP networks. It is an extension of xCHOKe, CHOKe and RED-PD schemes, combining both CHOKe hit and RED drop/mark histories, to detect, control and punish these flows more accurately while providing better protection to non-malicious flows. However, unlike xCHOKe and CHOKe, RECHOKe does not drop packets during CHOKe hits; thereby eliminating the complexity of dropping or marking randomly selected packets already queued and the unreliability of CHOKe hits. We analyze xCHOKe and RECHOKe in detail using ns-2 and show that RECHOKe performs better than RED, CHOKe and xCHOKe which are limited in what they can achieve as malicious flows get much more than their fair share and non-malicious flows get mistakenly penalized.

***Index Terms*—TCP Congestion Control, Congestion Avoidance, Active Queue Management (AQM), Buffer Management, Random Early Detection (RED)**

I. Introduction

TO control TCP-friendly flows from non-adaptive sources such as UDP and non-TCP friendly sources, we are proposing a new preferential dropping scheme called RECHOKe (REpeatedly CHOose and Keep for responsive flows, REpeatedly CHOose and Kill for unresponsive flows). It aims to improve on proposed schemes such as CHOKe [1] and its variants such as xCHOKe [1] for router buffer management.

CHOKe is a buffer management scheme that enables routers in an IP network to control congestion in the case when TCP segments are not the only segments queued. CHOKe compares each newly arriving packet with a randomly selected packet from the queue. If they are from the same flow (referred to as a CHOKE hit), then the both packets are dropped; otherwise the arriving packet is allowed to enter the queue (referred to as a CHOKE miss). CHOKe has been shown to be erratic in recognizing malicious flows [1] while often punishing non-malicious flows. Its advantage of being stateless can become a disadvantage since if it had kept some state, it would be able to make better and more accurate decisions. To improve upon CHOKe, xCHOKe was introduced. In xCHOKe, hits are stored in a table which is refreshed periodically (e.g., every t ms). xCHOKE uses this table to check whether the arriving packet's flow label is already in it. If it is (referred to as a table hit), the arriving packet is dropped or marked for dropping with a probability p^*. After this step the packet is compared with a randomly selected packet from the queue. If this results in a hit (CHOKe hit), then both packets are dropped or marked for dropping and the flow label is added to the lookup table (associated with a hit counter of one). If the flow is already in the table, the associated hit counter is incremented. The hit counter of flows in the lookup table is used to compute the probability p^*.

However, since xCHOKe is dependent on CHOKe hits to drop packets and update its table, it often punishes non-malicious flows as well. xCHOKe performs better than CHOKe by keeping partial state, however it (similarly to CHOKe) does not control malicious flows with bandwidths smaller than the link capacity. xCHOKe was shown to behave like CHOKe when the number of malicious flows is small. There are other schemes as well for malicious flow detection and punishment; the reader is referred to [3] for a more detailed discussion and description of these and other schemes.

Our scheme, RECHOKe, which combines ideas behind xCHOKe, CHOKe and RED-PD [5], succeeds in detecting, controlling and punishing malicious flows better and faster than schemes like xCHOKe and CHOKe. It does this by combining and using the CHOKe hit and CHOKe-RED drop/mark histories. We denote the RED queue used by the CHOKe scheme by affixing “RED” after the name of the CHOKe, thus we will be talking about CHOKe-RED queues. The basic idea behind this extension is the observation that in RED queues the choice of a particular flow to mark or drop during congestion is roughly proportional to that flow's share of the bandwidth at the router. RED-PD, on the other hand, requires proper values for their parameters such as the target RTT, K, and M. The overhead of using schemes like RED-PD

Manuscript received September 28, 2006.

Visvasuresh Victor Govindaswamy is with the University of Texas at Arlington, Arlington, TX 76013 USA (phone: 817-460-8487; e-mail: lovebat814@ yahoo.com).

Gergely Záruba is with the University of Texas at Arlington, Arlington, TX 76013 USA (e-mail: zaruba@ uta.edu).

G. Balasekaran is with the Nanyang Technology University, Singapore (e-mail: gbalas@nie.edu.sg).

in today's networks lies in that they are complex for the router to constantly calculate the RTT of a flow since RTT is heavily dependent on the ever-changing state of the network. For RECHOKe, we use both CHOKE hit and CHOKe-RED drop/mark histories to lower the number of false alarms and detect malicious flows faster than using one or the other. The simulations in this paper show that RED, CHOKe and xCHOKe are limited in what they can achieve since malicious flows still get significantly more than their fair share and non-malicious flows get mistakenly penalized. We also show the unreliability of RED, CHOKe and xCHOKe in protecting malicious flows. RECHOKe helps to isolate, control and punish malicious flows as it can be used with any active queue management scheme such as Random Early Detection (RED) [6].

In [3] we have conducted experiments to verify the accuracy of using both CHOKe hit and CHOKe-RED drop/mark histories in detecting malicious flows. Due to space constraints the reader is advised to refer to [3] for a more detailed analysis of RED, CHOKe, xCHOKE and RECHOKe.

II. RECHOKE

This section presents the functional description of RECHOKe. In RECHOKe, if the average queue length (*avq*) is smaller than a minimum threshold Th_{min}, then arriving packets are accepted. If *avq* is greater than the Th_{min} but smaller than a maximum threshold Th_{max}, a lookup table is queried to see whether the arriving packet's flow label is present there. If it indeed is (table hit), the arriving packet is marked for dropping and the flow's associated hit counter is incremented. A packet is then selected at random from the queue and its flow label is compared with that of the arriving packet. If the flow labels are the same (CHOKe hit), the flow label is added to the lookup table with an initial value of one for the associated hit counter. If the flow is already in the table, the associated hit counter is incremented. Unlike xCHOKe, the packets are not dropped or marked for dropping as a result of a CHOKe hit. They are allowed to enter the FIFO buffer. This removes some of the complexity of dropping the random packet after it has been admitted into the buffer and the unreliability of CHOKe hits. To remove this complexity from CHOKe, the authors in [1] proposed to add one extra bit to the packet header so that the bit is set to one if the random packet is to be dropped. When this packet advances to the head of the RED buffer, the status of this bit determines whether it is to be immediately discarded or transmitted on the outgoing line. We feel that this idea leads to a wasteful use of valuable buffer space and the addition of complexities due to the bit modification.

In the RED FIFO, if a packet is dropped or marked (in the presence of ECN [7]) by RED (called a RED hit), then its label is added to a lookup table with an initial value of one for an associated hit counter. If the flow is already in the table, the associated hit counter is incremented. If *avq* is greater than Th_{max}, the packet is dropped and its label is added to a lookup table with an initial hit counter value of 1. If the flow is already in the table (table hit), the associated hit counter is incremented. The hit counter *n* of an entry in the lookup table is used to compute the probability *p** and an indicator value *c** which are used to drop packets; these values are computed as follows:

$$p^* = \min(1, p_a \times 2^n) \text{ and } c^* = f\left(n \geq \frac{\sum_{i=1}^{m} n_i}{(2 \times m)} \right)$$

where p_a is RED's dropping probability and *m* is the number of entries in the table. If c* evaluates to true then the packet will be marked or dropped with probability p*. The purpose of c* is that we do not want to punish a TCP-friendly flow, whose previous packets just "accidentally" experienced a CHOKe hit leading to one of its recent packets experiencing a table hit. This scenario is common in CHOKe and xCHOKe, which results in TCP-friendly or responsive flows getting punished unnecessarily.

Hence, in RECHOKe, we update the table when an incoming packet has: i) hits in the lookup table (table hits), ii) hits with a randomly chosen packet from the buffer (CHOKe hits), or iii) it has been selected to be dropped or marked by RED (RED hits). The first two cases are similar to xCHOKe, while the third is similar to RED and RED-PD behavior. We can use RED drop/mark history in the presence of ECN[7].

In [3] detailed simulation results can be found on CHOKe hit, CHOKe miss and RED drop behavior. To show problems with CHOKe buffer management we have looked at two different slightly modified versions of CHOKe referred to as Half1 and Half2. In Half1 CHOKe, packets are randomly chosen from the first half of the buffer while in Half2 CHOKe from the bottom half. We have seen that depending on bandwidth required by malicious flows one of these approaches outperforms the other (and vice versa). RECHOKe starts off with the original CHOKe algorithm, sampling at certain intervals, say at time t=250ms, the first quarter of the buffer with the entries in the table. It computes the occupancy percentage at the first quarter of the occupied queue by flows with entries in the table. If this value is greater than 30%, then RECHOKe reverts to Half1 CHOKe, and to Half2 CHOKe otherwise.

In RECHOKe, we associate a time-to-live (TTL) with each entry in the lookup table. However, TTLs are not refreshed when an incoming packet has: i) table hits, ii) CHOKe hits, and iii) RED Hits. The first two cases again are similar to xCHOKe while the third case is unique to RECHOKe.

III. EVALUATION OF XCHOKE AND RECHOKE

In this section we analyze xCHOKe and RECHOKe buffers; to do this, we define the following terms. A table hit occurs when the flow id of the arriving packet matches a flow id entry already in the table. A table hit can be either good or bad. The table hit is good if the flow that the hit packet belongs to is malicious; it is a bad hit otherwise. Table drops are table hits that were selected for dropping by xCHOKe (after drawing a random with probability p*); for RECHOKe, these are table hits that were selected after evaluating p* and c*. Good table drops occur when the dropped packet belongs

to a malicious flow; bad table drops occur otherwise. A CHOKe miss victim is a packet which after experiencing a CHOKe miss enters the RED queue. CHOKe miss RED hit occurs when the flow id of the CHOKe miss victim matches the flow whose packets dominate the buffer at the time of selection of the CHOKe miss victim; CHOKe miss RED miss occur otherwise. A CHOKe miss RED hit could be either a good or bad: it is good if the victim packet belongs to a malicious flow; it is bad otherwise. A CHOKe miss RED miss again could be either a good or a bad CHOKe miss RED miss. Good CHOKe miss RED miss is when the victim packet belongs to a non-malicious flow; it is bad otherwise.

Using ns-2 [8], we simulate a bar-bell topology and analyze the xCHOKe and RECHOKe buffers in terms of buffer occupancy by a malicious flow (a constant-bit-rate UDP flow) competing with 10 TCP flows over a 3-Mbps link. All TCP flows have the same round trip propagation delay of 20ms with each output link having a latency of 1 ms and capacity of 10 Mbps. The parameters for the xCHOKe and RECHOKe buffers are: Th_{min} = 10, Th_{max} = 50, P_{max} = 0.1.

We ran simulations in 11 steps, starting with a malicious UDP rate of 0.5 Mbps, increasing the rate in 1 Mbps increments. Each simulation was long enough for initial transients to settle and be insignificant. Where appropriate, enough simulations were run to claim a 95% confidence that the confidence intervals are less than 5% of the mean.

Fig 1 shows the number of table hits for both the xCHOKe and RECHOKe algorithms. The number of hits with RECHOKe is 50-65% greater than with xCHOKe. The combination of CHOKe and CHOKe-RED histories allow RECHOKe to register a large number of hits and using these hits, RECHOKe can isolate malicious flows faster and more accurately. RECHOKe does punish malicious flows more rigorously while not penalizing TCP-friendly flows as it can be observed by comparing Figs 18 and 19. The majority of the hits shown in Fig 1 are good table hits as witnessed by Figs 2 and 3, pointing to RECHOKe identifying malicious flows better than xCHOKe. Fig 2 also shows that the number of good table hits for RECHOKe is 50-65% greater than that of xCHOKe; Fig 3 shows that the number of bad table hits for RECHOKe is 2-16 times less as well and constant no matter what the rate of the UDP flow is.

Next, we analyze the number of table drops for both algorithms. Ideally drops should only consist of good table drops. Fig 4 shows the number of table drops for both algorithms. Fig 5 shows the number of good table drops; we can observe that RECHOKe outperforms xCHOKe significantly. In Fig 6 we can also observe that the same is true in terms of bad table drops. Nearly all the table misses (not shown here) for both algorithms are good table misses. We can conclude that RECHOKe drops fewer packets from TCP- friendly flows than xCHOKe. Due to the large number of good table drops and the almost insignificant number of bad table

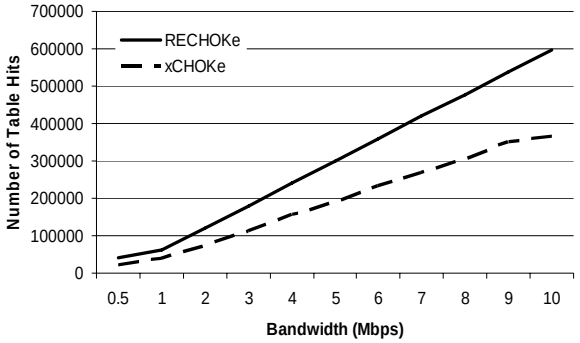

Figure 1. The number of table hits vs. UDP rate (RECHOKe and xCHOKe).

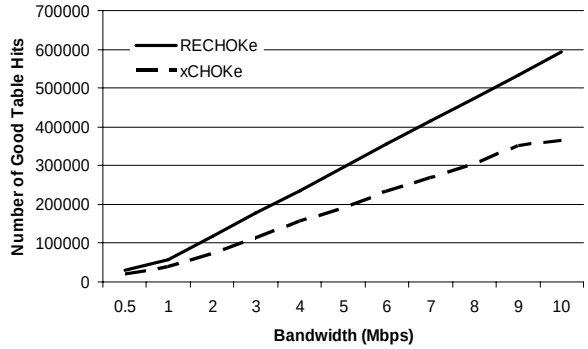

Figure 2. The number of good table hits vs. UDP rate (RECHOKe and xCHOKe).

drops, RECHOKe is faster and more accurate in the identification of flows. Unlike xCHOKe, RECHOKe updates the table after table hits. This is necessary as with the lifetime of flows, CHOKe hits become more unreliable since the number of packets belonging to the malicious flow in the buffer are reduced drastically leading to a greater number of bad CHOKe hits (see Figs 9 and 10). The same phenomenon can be observed in xCHOKe and CHOKe, which leads to unnecessary punishment of TCP-friendly flows. Figs 7 and 8 show that the number of CHOKe hits and misses for both algorithms, reflect this trend. CHOKe hits did not affect RECHOKe as the packets responsible for these hits were not dropped but passed on to the RED buffer. Although these hits were updated in the table, the c* condition prevents unnecessary drops of TCP-friendly packets.

Upon analyzing the CHOKe misses in Figs 11 and 12, we have found that RECHOKe significantly outperforms xCHOKe in bad CHOKe misses and good CHOKe misses despite having the larger number of CHOKe misses. This shows that RECHOKe allows packets from TCP-friendly flows to enter the RED buffer more frequently than xCHOKe.

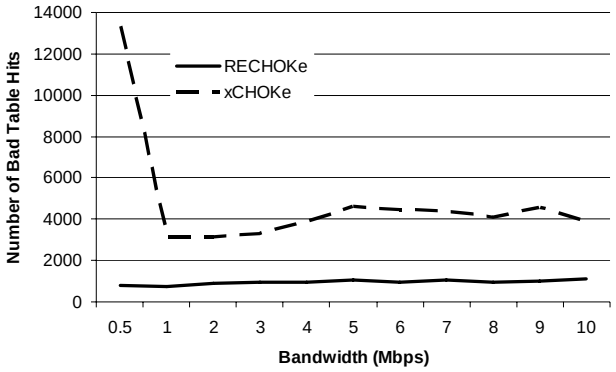

Figure 3. The number of bad table hits vs. UDP rate (RECHOKe and xCHOKe)

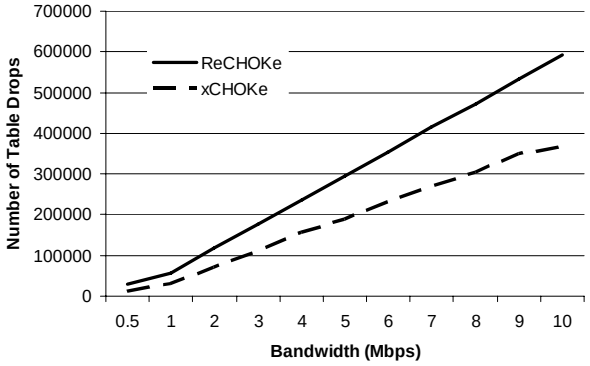

Figure 4. The number of table drops vs. UDP rate (RECHOKe and xCHOKe).

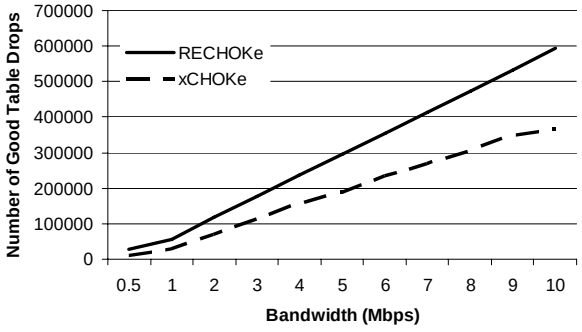

Figure 5. The number of good table drops vs. UDP rate (RECHOKe and xCHOKe).

Next, we analyze RED hits and misses for both RECHOKe and xCHOKe. Fig 13 shows that the number of dropped packets is greater when using RECHOKe. This is true because during CHOKe hits, packets are not dropped but admitted into the buffer; as a result, more packets need to be dropped by the RED buffer. Analyzing Figs 14 and 15 and we can see that most of the RED hits with RECHOKe are good compared to those of xCHOKe, i.e., most dropped packets belonged to malicious flows. This happens in spite of most of the admitted packets being from the ten TCP friendly flows. After the good RED hits, the malicious flows became no longer dominant. Hence, RED took out more UDP packets from (the already the few) that had been admitted into the buffer.

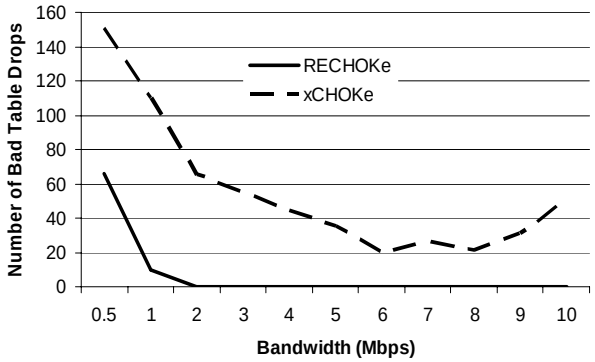

Figure 6. The number of bad table drops vs. UDP rate (RECHOKe and xCHOKe).

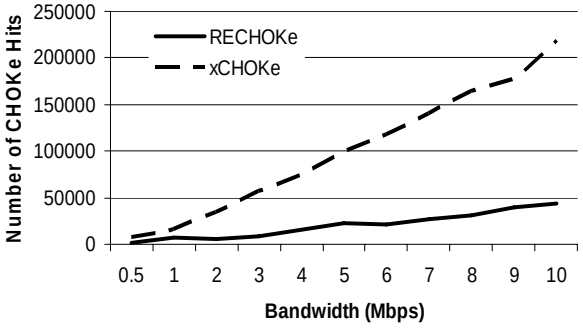

Figure 7. The number of CHOKe hits vs. UDP rate (RECHOKe and xCHOKe).

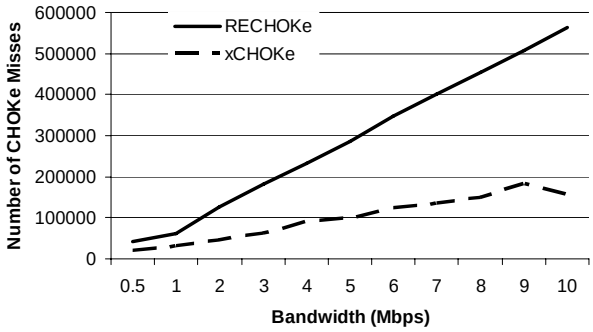

Figure 8. The number of CHOKe misses vs. UDP rate (RECHOKe and xCHOKe).

Next, we introduce two UDP flows: UDP-1 is set at 1 Mbps and UDP-2 at 3 Mbps with the 10 TCP flows varied at 2 ms delay increments starting at 20ms (TCP-1). We use the same bar-bell topology with the same RED parameters as before.

Figs 16, 17, 18 and 19 show the bandwidth allocation for the congested link for each flow for RED, CHOKe, xCHOKe and RECHOKe respectively. RED does not control the malicious flows (UDP-1 and UDP-2) against gaining control of almost the entire link bandwidth. The bandwidths of the

TCP-friendly flows (TCP-1 to TCP-10) were almost zero. CHOKe and xCHOKe did control them more than RED did and during the process of controlling these malicious flows, (due to the fact that TCP flows suffering CHOKe hits [3]) they suffered bandwidth losses which in turn, were used by the very flows that these algorithms were trying to control. By minimizing bad CHOKe hits, RECHOKe controls malicious flows better than CHOKe and xCHOKe. RECHOKe also performed best in protecting TCP-friendly flows. Fig 20 shows the UDP flows at RECHOKe, xCHOKe, CHOKe and RED buffers.

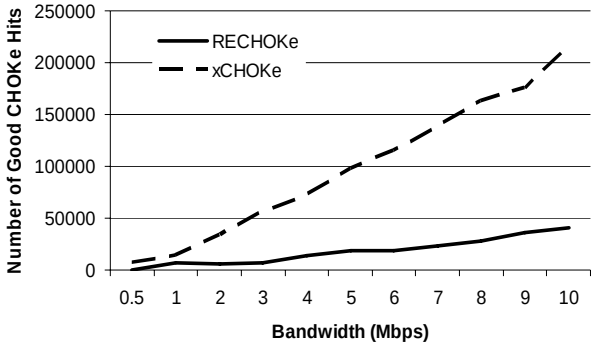

Figure 9. The number of good CHOKe hits vs. UDP rate (RECHOKe and xCHOKe).

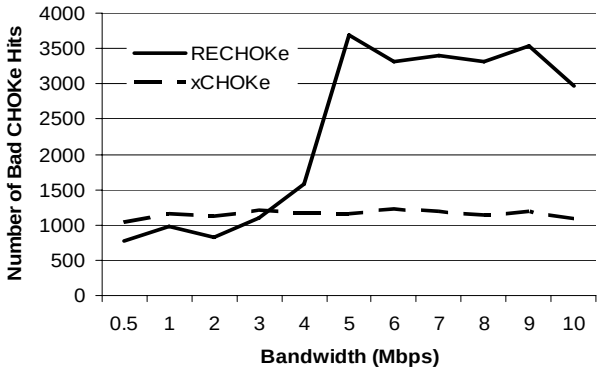

Figure 10. The number of bad CHOKe hits vs. UDP rate.

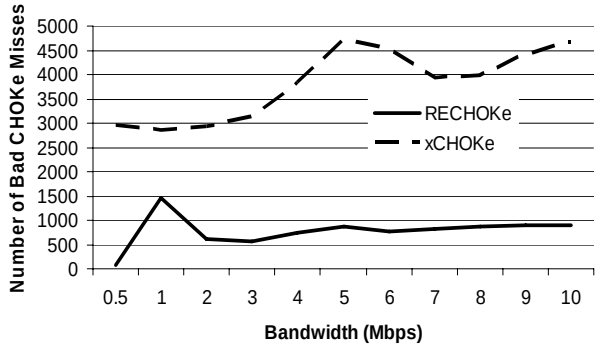

Figure 11. The number of bad CHOKe misses vs. UDP rate (RECHOKe and xCHOKe).

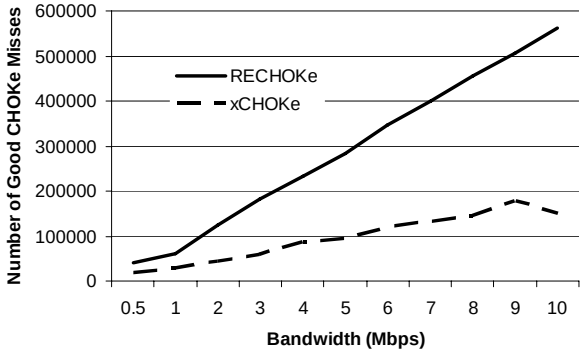

Figure 12. The number of good CHOKe misses vs. UDP rate.

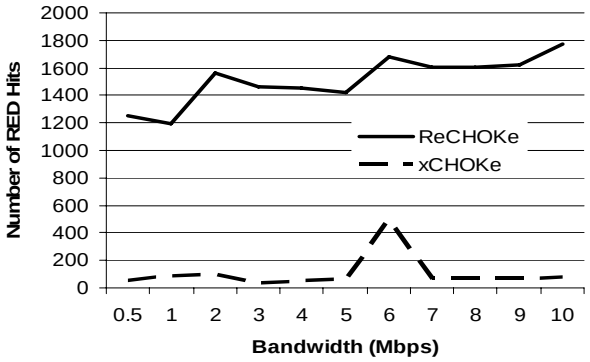

Figure 13. The number of RED hits vs. UDP rate.

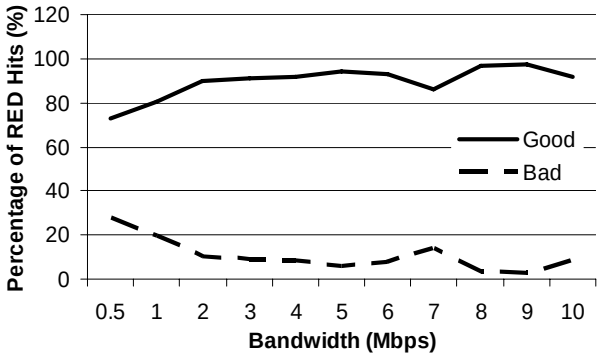

Figure 14. The percentages of good and bad RED hits vs. UDP rate (RECHOKe).

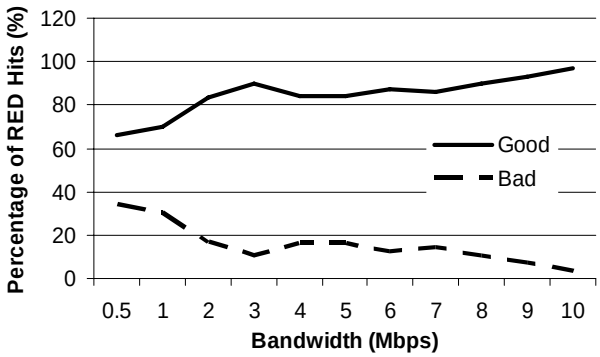

Figure 15. The percentages of good and bad RED hits vs. UDP rate (xCHOKe).

IV. Conclusion and Future Work

In this paper, we have presented RECHOKe, a scheme for detecting, controlling and punishing malicious flows in IP networks. We showed that RECHOKe outperforms RED, CHOKe and xCHOKe by combining the techniques used by xCHOKe and RED-PD in identifying and punishing malicious flows while eliminating the complexity of dropping or marking randomly selected packets already queued (a method used by both CHOKe and xCHOKe) and the unreliability of CHOKe hits.

We are currently working on implementing RECHOKe in Linux based packet routers and by using various flavors of TCP with multi-bottleneck topologies. We are also presently working on a mathematical model for RECHOKe.

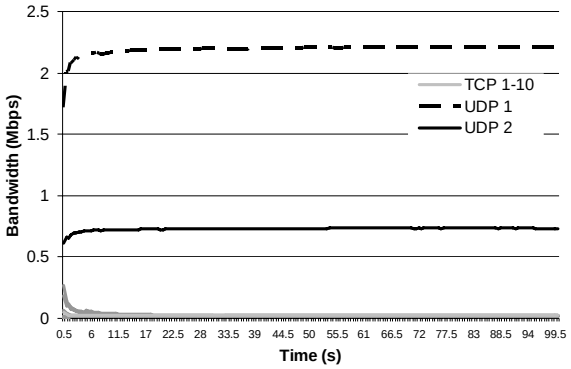

Figure 16. Link utilization at the RED buffer

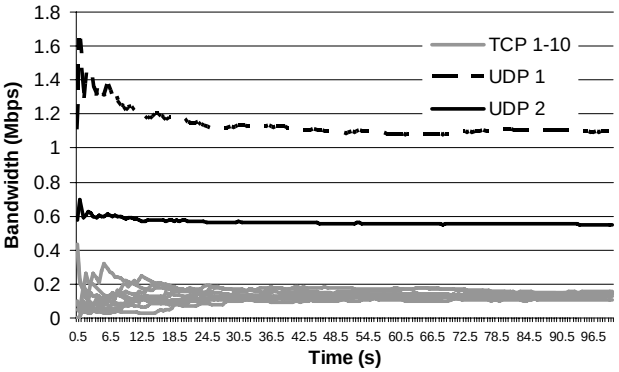

Figure 17. Link utilization at the CHOKe buffer

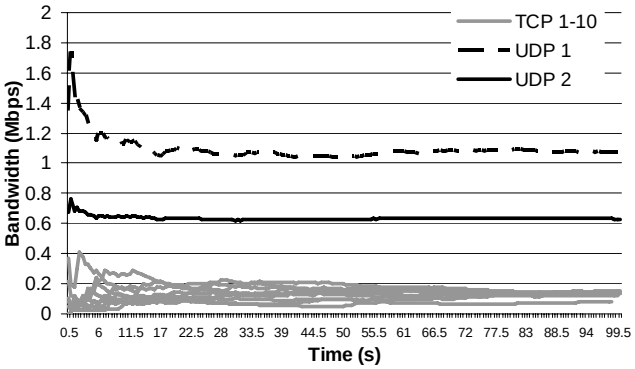

Figure 18. Link utilization at the xCHOKe buffer

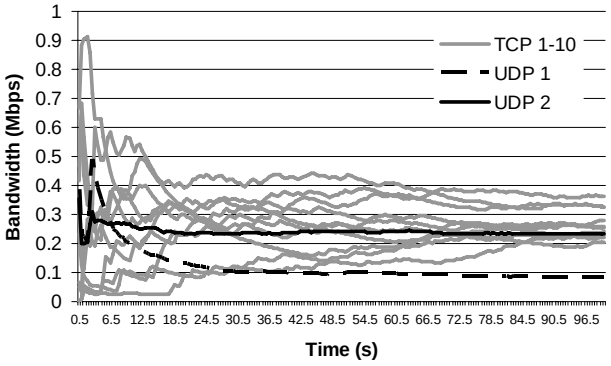

Figure 19. Link utilization at the RECHOKe buffer

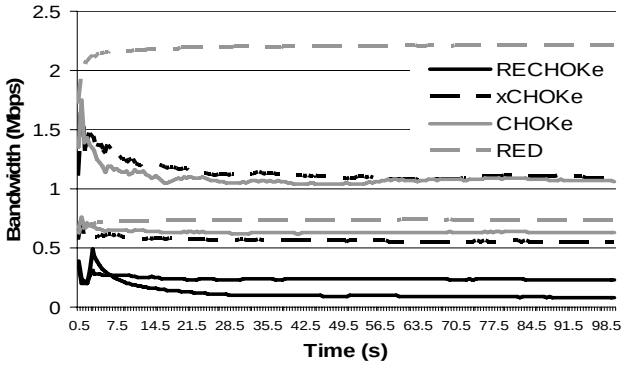

Figure 20. UDP flows at RECHOKe, xCHOKe, CHOKe and RED buffers

References

[1] R. Pan, B. Prabhakar, and K. Psounis, "CHOKe: a stateless active queue management scheme for approximating fair bandwidth allocation," *Proc. of IEEE INFOCOM*, Tel Aviv, Israel, March 2000, vol. 2, pp. 942–951.

[2] P. Chhabra, A. John, H. Saran, and R. Shorey. "Controlling Malicious Sources at Internet Gateways," *IEEE Int. Conf. Communication*, Anchorage, Alaska, May 2003, vol. 3, pp. 1636-1640.

[3] V. V. Govindaswamy, G Záruba and G. Balasekaran, "RECHOKe and RCUBE," *UTA Technical Report 2006-7*, September, 2006. http://omega.uta.edu/~victor/researchFrame.htm

[4] V.V. Govindaswamy, G. Zaruba, G. Balasekaran, "Analyzing the Accuracy of CHOKe Hits, CHOKe Misses and CHOKe-RED Drops", Submitted for Publication-2007, details pending.

[5] R. Mahajan, S. Floyd and D. Wetherall, "Controlling high-bandwidth flows at the congested router," *9th Int. Conf. Network Protocols*, Nov. 2001, pp. 192-201.

[6] S. Floyd and V. Jacobson, "Random early detection gateways for congestion avoidance," *IEEE/ACM Trans. Networking*, August, 1993, vol. 1, no. 4, pp. 397-413.

[7] K. Ramakrishnan, S. Floyd and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," *RFC 3168*, September, 2001.

[8] NS2 simulator. Available: http://www.isi.edu/nsnam/ns/.