Implementing Pushback: Router-Based Defense Against DDoS Attacks

John Ioannidis
ji@research.att.com
AT&T Labs Research

Steven M. Bellovin
smb@research.att.com
AT&T Labs Research

Abstract

Pushback is a mechanism for defending against distributed denial-of-service (DDoS) attacks. DDoS attacks are treated as a congestion-control problem, but because most such congestion is caused by malicious hosts not obeying traditional end-to-end congestion control, the problem must be handled by the routers. Functionality is added to each router to detect and preferentially drop packets that probably belong to an attack. Upstream routers are also notified to drop such packets (hence the term Pushback*) in order that the router's resources be used to route legitimate traffic. In this paper we present an architecture for Pushback, its implementation under FreeBSD, and suggestions for how such a system can be implemented in core routers.*

1 Introduction

Distributed Denial of Service (DDoS) attacks have become an increasingly frequent disturbance of the global Internet[MVS01]. They are very hard to defend against because they do not target specific vulnerabilities of systems, but rather the very fact that the target is connected to the network. All known DDoS attacks take advantage of the large number of hosts on the Internet that have poor or no security; the perpetrators break into such hosts, install slave programs, and at the right time instruct thousands of these slave programs to attack a particular destination. The attack does not have to exploit a security hole at the target to cause a problem (although that would exacerbate the problem, to the attackers' benefit).

Unlike most security attacks, there is almost nothing the victim can do to protect itself. What is being attacked is usually not a specific vulnerability, but rather the very fact that the victim is connected to the network. Under normal operating conditions, and assuming that its link(s) and processing capacity have been adequately provisioned, the standard, TCP-like congestion control ensures fair use of the available resources. Under a DDoS attack, the arriving packets do not obey end-to-end congestion control algorithms; rather, they incessantly bombard the victim, causing the well-behaved flows to back off and eventually starve. In addition, a large-scale DDoS attack not only causes trouble to its intended victim, but also interferes with other traffic that may happen to share a portion of the network that is being heavily congested.

Mahajan *et al.* [MBF$^+$a, MBF$^+$b] introduce a network-based solution, called Pushback, to address the question of whether anything can be done *inside* the network to defend against DDoS attacks, and evaluate the solution with extensive simulations. In this paper, we present an implementation of these concepts under Unix, along with experimental results from our laboratory testbed. In the rest of this section we give a very brief overview of the Pushback mechanism; the reader should refer to [MBF$^+$a] for all the details. Section 2 presents the architecture of a router that can support Pushback; Section 3 gives implementation and performance details; we conclude with a discussion of deployment options, as well as related work.

1.1 Overview of Pushback

If we could unequivocally detect packets belonging to an attack and drop just those, the problem would be solved. However, routers cannot tell with total certainty whether a packet actually belongs to a 'good' or a 'bad' flow; our goal will be to develop heuristics that try to identify most of the bad packets, while trying not to interfere with the good ones. Again, Mahajan *et al.* introduce the concept of *Aggregate-based Congestion Control (ACC)*; in this context, an *aggregate* is defined as a subset of the traffic with an identifiable property. For example, "packets to destination D", "TCP SYN packets", or even "IP packets with a bad checksum" are all potential descriptions of aggregates. The task is to identify aggregates responsible for congestion, and preferentially drop them at the routers.

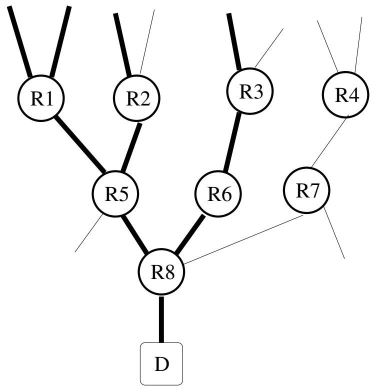

Figure 1. A DDoS attack in progress.

To illustrate Pushback, consider the network in Figure 1. The server D is under attack; the routers R_n are the last few routers by which traffic reaches D. The thick lines show links through which attack traffic is flowing; the thin lines show links with no bad traffic. Only the last link is actually congested, as the inner part of the network is adequately provisioned. In the absence of any special measures, hardly any non-attack traffic would be reaching the destination. Some non-attack traffic is flowing through the links between R2-R5, R3-R6 R5-R8, R6-R8, and from R8 to D, but most of it is dropped due to congestion in R8-D. Throughout this paper we shall be referring to 'good', 'bad', and 'poor' traffic and packets. *Bad* packets are those sent by the attackers. Bad traffic is characterized by an *attack signature*, which we strive to identify; what can be really identified is the *congestion signature*, which is the set of properties of the aggregate identified as causing problems. *Poor* traffic consists of packets that match the congestion signature, but are not really part of an attack; they are just unlucky enough to have the same destination, or some other properties that cause them to be identified as belonging to the attack. *Good* traffic does not match the congestion signature, but shares links with the bad traffic and may thus suffer.

In the figure above, some of the traffic entering R4 is good (the part exiting R7 that is not going to R8), and some is poor, as it is going to D. There may be some good traffic entering R5 from the links above, and exiting from the lower left link, but depending on how congested the links R1-R5 and R2-R5 are, it may suffer. The other links have a mixture of bad and poor traffic. Now, no matter how smart filters R8 could employ, it cannot do anything to allow more good traffic originating from the left side of the graph to reach D. All it can do is preferentially drop traffic arriving from R5 and R6, hoping that more good traffic would flow in via R7. With Pushback, R8 sends messages to R5 and R6 telling *them* to rate-limit traffic for D. Even though the links downstream from R5 and R6 are not congested, when packets arrive at R8 they are going to be dropped anyway, so they may as well be dropped at R5 and R6. These two routers, in turn, propagate the request up to R1, R2, and R3, telling *them* to rate-limit the bad traffic, allowing some of the 'poor' traffic, and more of the good traffic, to flow through.

In the rest of the paper, we shall describe a router architecture and the corresponding algorithm that can implement this kind of defense, and present its implementation under FreeBSD.

2 Architecture

Consider a typical router; Figure 2 gives the view of the routing mechanism from one output interface. There are several incoming links, and the routing subsystem is implicitly shown in the choice of the output interface. A rate limiter is introduced before the output queue; some form of rate limiting or traffic shaping is already in place in many routers. For example, in the FreeBSD operating system [FBS], the IPFW firewall package also does traffic shaping. The simplest way to view the rate limiter is as a predicate that decides whether a packet is dropped or forwarded. In our architecture, dropped packets are sent to the pushback daemon, `pushbackd`. The daemon, in turn, periodically updates the parameters of the rate limiter, and also informs the upstream daemons to update theirs. It is interesting to note that the actual Pushback daemon may not reside on the router itself, but rather on an external ancillary piece of equipment.

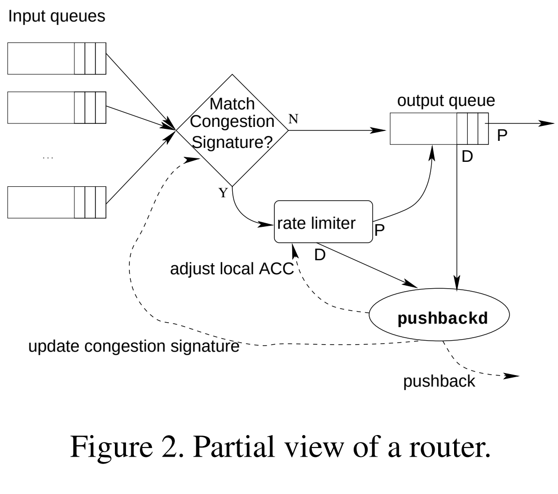

Figure 2. Partial view of a router.

The information sent to the Pushback daemon by the rate limiter is shown in Figure 3. Most of the fields have the obvious purpose. The magic number provides some protection against synchronization problems between the kernel and the user-level process. The timestamp is expressed in nanoseconds since the router was last booted, and its purpose, along with the packet size, is to allow the code to estimate the bandwidth that would have been consumed by the dropped packets. The ‘reason’ field indicates whether this was a tail-queue drop, a RED drop, and so on. Only packets dropped because of queue discipline restrictions are logged; packets dropped because, for example, they were not routable, or even because no buffer space could be allocated for them at the driver may not even reach this part of the code, so they are not reported at all.

Magic number
IP Destination address
Input interface
Output interface
Timestamp
Packet size
Reason

Figure 3. **Dropped packet report**

It is important to note that the design decision to separate the rate-limiting and packet-dropping functionality from the rest of the pushback mechanism has implications for the eventual deployment of such a mechanism in the Internet. Routers can be designed[1] to report information about dropped packets, either to a process running on the router CPU, or a computer attached to the router using a local interface. All the intelligence, which would have to evolve rapidly as DDoS attacks change in nature, would reside in easy to replace, generic PCs, and scarce router resources do not have to be allocated to the Pushback task.

2.1 Aggregate Detection

Periodically, `pushbackd` processes the saved drop-set to try to detect congestion. A large amount of dropped packets obviously indicates congestion. It now has to determine if there is an attack going on, and whether to respond to it. The exact algorithm(s) to run are will be an important research topic for some time to come. We present such an algorithm here; another can be found in [MBF$^+$a].

We start by considering the drop set, that is, the set of packets that are dropped by the rate limiter. These may include packets dropped because of already-existing pushback activity, but as we shall see, this does not affect the outcome of the algorithm. The size of the drop set should be large enough to allow meaningful results, but also small enough to be processed in a small amount of time. Fortunately, it is not necessary to keep all the dropped packets; a representative sample of them suffices, as it is the most frequent packets that will affect the outcome of the algorithm, and these will be the most represented ones in the sample. The important feature is that the algorithm should run in less time that it takes to collect the packets.

This algorithm detects aggregates based only on IP destination address; the assumptions are that source addresses cannot be trusted anyway, and that the attackers are just sending IP packets with random contents to attempt to congest the target link. It starts by deciding whether the congestion level is high enough, that is, the drop rate is high enough, to warrant attempting to do preferential dropping. A simple test is whether the bandwidth w_o of the output link would be exceeded by more than an acceptable drop rate, say, 20% of the traffic, in other words, if $w_i > 1.2 \times w_o$, where w_i is

[1]in fact, some high-end routers already do that

the total incoming bandwidth from all input links. If this is the case, the algorithm starts by matching the destination address of each dropped packet against the routing table, and selecting the longest matching prefix. This groups the dropped packets according to their eventual destination link in some downstream router (or even the target, if the target has multiple IP addresses). The drop set is then sorted with the prefix as the key. We now want to find the prefix with the highest count. This can be done in a single pass. The sorting has complexity $O(n \log n)$, where n is the size of the drop set. The counting and determination of the most frequent prefix is, of course, linear in n. Now, for the subset of the drop set that matches this selected prefix, we perform another scan to see if the destinations of the dropped packets match a longer prefix than the routing prefix. If, for example, a single machine has been targeted, even though the prefix garnered from the routing table will be shorter than 32 (or 128, in the case of IPv6) bits, the address of the selected aggregate will be the full 32 (or 128) bits. The selected prefix constitutes the congestion signature.

It is likely that more than one attack is happening at the same time. To determine that, the drop set is scanned to compute the fraction of the bandwidth w_b for which the identified congestion signature is responsible. If removing the traffic identified the congestion signature does not bring the output traffic below the previously described acceptable level, that is, if $w_i - w_b > 1.2 \times w_o$ still holds, the algorithm is repeated in the hope of adding more prefixes to the congestion signatures. The precise parameters would be tunable to specific installations.

In some case we may not be able to find a second prefix (or even a first) responsible for a significant fraction of the traffic; that would be the case where congestion is not caused by an attack, or traffic to a specific destination, but by a general increase in background traffic. We then rate-limit what we can, and let the queue management of the output link, whatever it is, handle the rest of the congestion.

2.2 Rate Limiting

Once the congestion signature has been identified, the code must decide what to rate-limit it to. If $w_b > w_l$, where $w_l = w_i - 1.2 \times w_o$, then we simply rate-limit the aggregate down to w_l, and pass the rest of the traffic on. If $w_b < w_l$, we eliminate all traffic belonging to the congestion signature, and let the rest of the excess traffic be limited by the output queue. Note that the traffic that passes through the rate limiter and is not dropped is not treated preferentially; it is also sent to the output queue of the interface and treated just like the rest of the output traffic. That is, just because a packet was not dropped by the rate limiter does not mean that it will be preferentially treated and not dropped if it must at the output queue.

The pushback daemon gets dropped packets from both the rate limiter and the output queue; if the rate of the attack stays constant, the daemon will get the same number of dropped packets, but it will be getting more of them from the rate limiter. This means that, as long as the attack is going on, the pushback daemon will keep telling the rate limiter to drop packets; when it stops, no special action needs to be taken; at the next update, no attack will be identified, and the rate limiting will stop. Naturally, some damping may be necessary to avoid oscillations; experience with these mechanisms in production networks will be necessary before all the details can be worked out.

2.3 Pushback

So far we have described the *local* version of ACC. This is not enough, however. Once the Pushback daemon has identified a prefix to rate-limit, it communicates that information to its upstream links. The messages exchanged by routers implementing Pushback are described in detail in [FBI$^+$]. There are three such messages: request, response, and status. The pushback request is shown in Figure 4.

Various header fields
RLS-ID
Maximum depth
Depth of Requesting Node
Bandwidth Limit
Expiration Time
Congestion Signature

Figure 4. **Pushback request**

Each request has a Rate-Limiting Session Identifier (RLS-ID), which is used to match responses to requests. The Depth field is used to set a limit to the propagation of the pushback requests. The depth of the originator is 0; upstream daemons add 1 to the depth before propagating the message. The maximum depth of propagation is set by the originating router and passed along by each subsequent router. Pushback uses soft state; there is no explicit revocation of a pushback request, and no effort is made to recapture the state after a router reset. The expiration time is used to manage this soft state – if a Refresh message does not arrive before the expiration time has elapsed, the entry is deleted. Finally, the congestion signature is a list of destination prefixes that the bandwidth limit applies to.

Authentication of Pushback requests is an obvious concern. If the routers participating in Pushback are neighbors, simply sending the request out with a TTL (or hop count for IPv6) of 255 is sufficient; any request coming from an attacker would have a lower TTL, and if an attacker has compromised a router, there are more serious concerns to be resolved.

A special type of request is a cancel message, which instructs the upstream router to stop rate-limiting. It is useful when long expiration times are specified by default, and the originating router has decided that it no longer needs protection from upstream.

The Pushback daemon not only sends requests, but it also listens for requests from its downstream routers. Once it receives a pushback request (or refresh), it adds the appropriate rule to the rate limiter, and keeps track of the dropped packets it gets from it.

In addition to requests sent upstream, the pushback daemon also sends status messages downstream. These status messages contain a depth field; if it is non-zero, the response is simply passed along downstream (remember that pushback request messages are passed only between adjacent routers, possibly using non-globally-routable addresses; hence, the status messages have to follow the same hop-by-hop paths). Since many pushback operations may be happening at the same time, the downstream direction for any particular one can be determined by examining the local set to match the congestion signature and RLS-ID, and determine which interface it pertains to. Before propagating a request downstream, 1 is subtracted from the node depth.

When a daemon receives a response (with depth 0), it can use the information in it to determine whether to continue the pushback, or how to modify it. For example, it may determine that a larger fraction of traffic that was was requested is being dropped on the subtree upstream from one link, and much less traffic from the subtrees upstream from the rest of the links. Such information may be used to dynamically adapt the requests to match the evolving traffic patterns.

In the following section, we describe how this architecture is actually implemented under FreeBSD.

3 Implementation

Encouraged by the simulation results in [MBF$^+$a, MBF$^+$b], we implemented Pushback under FreeBSD. The tests were carried out on the network shown in Figure 5.

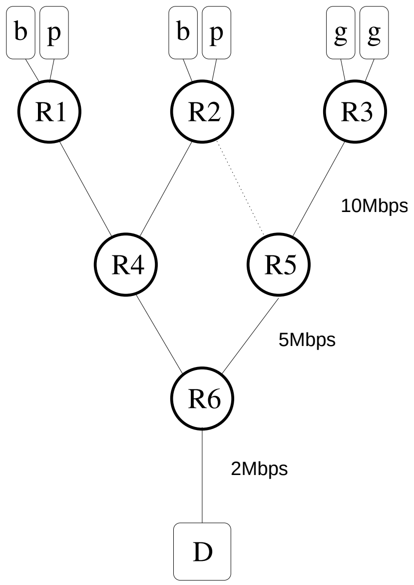

Figure 5. Testbed network.

The sources of bad (attack), poor (legitimate but sharing a congestion signature), and good (just legitimate) traffic are shown as *b, p,* and g, respectively. The destination of all traffic is *D*. *Rn* are routers; the links between the routers are 10Mbps for the first layer, 5Mbps for next layer, and 2Mbps between the last router and the destination. The test hardware being used (300MHz PentiumII generic PCs with 100Mbps Ethernet interfaces, running FreeBSD 4.2) can easily route at least 30Mbps of aggregate traffic without dropping any packets; in our experiments, substantially slower rates were used, both to give a margin of safety, and to make the actual measurements easier.

We use the IPFW packet filter functionality present in FreeBSD to both implement links of particular bandwidth-delay characteristics, and to implement the rate limiter. For, example the following two commands specify that the link outgoing on interface `xl2` has a bandwidth of 10 megabits per second, and its queue can hold 50 packets (which is the usual queue length of Ethernet interfaces).

```
# ipfw add 999 pipe 999
    ip from any to any
    out xmit xl2
# ipfw pipe 999
    config bw 10Mbits/s
    queue 50packets
```

The link speeds are configured by setting up the output speeds of the corresponding interfaces. No input speeds need to be configured. The rate limiting code in IPFW may apply various queue disciplines, such as RED; the default discipline is FIFO, and it is what is used in our experiments.

The kernel has been modified so that when a packet is dropped in the queue management code, the information described in Figure 3 is sent to `pushbackd`. Any number of mechanisms can be used to pass information from the kernel to the user; we chose to use the tunnel interface driver (`tun(4)`). When the output routine of the tunnel driver (`tunoutput()`) is called by the networking code in the kernel, the packet can be read from the user level by reading the corresponding `/dev/tun`*n* device. We use `/dev/tun63`. The user-level daemon, `pushbackd`, may keep only a sample of the dropped packets. The probability of keeping a packet may be inversely proportional to its size, so that a constant fraction of the bandwidth, rather than the packet count, is kept. At the traffic levels in the experiment, the code has no trouble keeping up. `Pushbackd` samples the dropped packets it receives, and periodically (every 60 seconds) runs the aggregate-detection algorithm described in Section 2.1. Even for a drop set of 10^5 packets, the algorithm runs in well under a second.

For each prefix that the daemon is rate-limiting, whether it is because of locally-detected congestion, or because of a Pushback request, it keeps an entry with all the fields shown in Figure 4, plus a starting time, in a linked list, ordered by expiration time. As an implementation optimization, only the difference between expiration times is kept in the corresponding fields, so that for each clock tick, only the first one has to be decremented. When the detection algorithm finds a prefix to rate-limit, it searches the list; if it is already there, and it is a prefix that was originated with the current node, it updates the expiration time, potentially moving it further back in the linked list.

When R6 detects congestion on its outgoing link to D, it runs the aggregate-detection algorithm, and decides that the target is 10.102.0.31. It then examines the traffic for D it is getting from R4 and R5, and it finds that R4 is sending it 5Mbps, but no traffic for D is arriving from R5 (the ‘g’ traffic coming in from R3 may be for another host in D’s subnet). The outgoing link is only 2Mbps, it is going to request R4 to limit the traffic they are sending to 2.4Mbps (20% more than the output rate, so that some traffic. When this happens R4 inserts a firewall rule such as this:

```
# ipfw add 101 pipe 101 ip
  from any to 10.102.0.31/32
  out xmit xl2
# ipfw pipe 101 config bw 2400Kbps
```

Since there is no traffic coming in from R5 for D, no pushback is sent to R5. Note that already we have a better situation; only 2.4Mbps are arriving from R4; therefore, if the ‘g’ sources upstream from R5 are sending traffic comparable to that, more of it will get through than before R4 started rate-limiting. However, this could have been accomplished by clever input filtering on R6, so let us proceed one step further.

R4 will now tell R1 and R2 to rate-limit traffic to D. In a similar fashion, more of the bad traffic from the ‘b’ sources entering each of R1 and R2 will be dropped, and some of the traffic from the ‘p’ (poor) sources destined for D makes it to D. Initial measurements indicate that Pushback is rate-limiting successfully, so that more of the poor and good traffic are getting through. Further study is needed to fine-tune the sampling rates, detection intervals, and also consider feedback information in adjusting the Pushback parameters.

It was no surpise that the system worked. The most interesting observation is that even though the hardware used is fairly old (a benefit in this case, as it is more easily overloaded), there was no noticeable system performance degradation. This also does not come as a surprise, since all the switching in the kernel is done in software and in the same address space. In a real router with hardware-assisted fast switching paths for the common cases, the overhead of imposing a number of rate limiting sessions may be much higher.

4 Related Work

Distributed Denial of Service attacks have been a real problem for less than three years, and not much published work exists on the subject. Related work falls into two categories: old work that can also be used in countering DDoS attacks, and new work specifically aimed at this task.

Originally, it was suggested that DDoS attacks could be countered by applying resource allocation techniques on network bandwidth. Integrated Services [CSZ92] and Differentiated Services [BBC$^+$98] are two approaches aimed at isolating flows with specific quality of service (QoS) requirements from lower-priority traffic. It is not clear if this approach would help; Web traffic, which is a significant fraction of network traffic, is likely to remain best-effort, so it will not be protected by QoS requirements. It is also not clear to what extent compromised sources could fake traffic to show it belonged to QoS-protected flows. There is also an approach that is similar to pushback that was described in [Van97] in an Active-Networks-based defense against flooding attacks.

There are many congestion-control mechanisms, which might alleviate some of the effects of congestion due to DDoS attacks if only they were globally deployed. Random Early Detect (RED)[FJ93] and its variants tries to identify flows that do not obey TCP-friendly end-to-end congestion control, and preferentially drop them. There is also a large body of work (*e.g.*, Fair Queuing[DKS89], Class-Based Queuing[FJ95]) aimed at allocating specific fractions of the available bandwidth to each flow so that they all get served. The main problem with these approaches is that packets belonging to DDoS attacks do not have readily-identifiable flow signatures, and can thus not be identified by these mechanisms. This is the reason why the concept of Aggregate-based Congestion Control was developed[MBF$^+$c] and which is where the work in the present paper is based.

A different style of approach to combating DDoS attacks focuses on trying to detect a DDoS attack in progress and then respond to the specific attack. Various forms of packet tracking have been suggested. Some try to construct a partial map of the paths that the attack is taking; the two main variants are *traceback* [Bel00] and packet marking [SWKA00, DFS01]. Others, such as [Sto00] involve all edge routers logging (a sample of all) packets, and then analyzing the logs to identify, and hopefully block, sources of the attack. An interesting approach is that of [BC00] where, starting with a partial map of the network, they can locate routers where attack traffic is coming from by flooding them and watching the attack traffic decrease. In traceback techniques, routers pick outgoing packets with a small probability (*e.g.* $1/20000$), and send a traceback packet (a new kind of ICMP packet) to the same destination as the sampled packet. The traceback packet contains the IP address of the router sending it, and is always sent out with a TTL (or Hop Limit) of 255 (as a rudimentary form of authentication). During an attack, a sufficient number of these packets will reach the target for it to form an approximate map of the path the attack is taking. Of course, the attacker can inject its own traceback packets, but unless it has compromised a router fairly close to the target (to inject packets with a sufficiently high TTL), all it will achieve is create a fictitious subtree from where the attack may be originating. Packet marking does not send additional packets, but rather modifies the IP ID field in each packet to carry partial information about the router that marked the packet. Clever techniques are used to reconstruct the original data. This approach has its own set of failures (*e.g.*, it is hard to get it to work with IPv6). The question of what to do even when the paths the attack is taking are identified still remains, and it may still involve human operators, especially when administrative boundaries are crossed.

The common problem that all the tracking techniques are trying to solve is that source addresses in attack packets cannot be trusted, because they are very easy to forge. If all edge routers in the entire

Internet were implementing source address filtering[FS98], this task would be greatly simplified. Of course, most machines where the packets are originating have been compromised by an attacker, and their owners do not even know that they are being used for an attack. Also, even if the hundreds or thousands of machines that an attack is coming from were known, it is not clear what could be done about them. Finally, it has been suggested[SDS00] that intrusion detection systems or firewalls be used to detect an attack in progress, and notify upstream elements accordingly.

We view Aggregate-based Congestion Control and Pushback as complimentary to many of these approaches. For example, a good map of the network with reliable historical traffic profiles from traces, can be used to determine sudden changes in traffic profiles that could signal an attack, or help determine how to allocate rate limits in pushback messages.

5 Discussion

Let us discuss some issues that may affect the way Pushback could be deployed. First off, it is fairly obvious that the pushback is most effective when an attack is non-isotropic; in other words, there will be routers fairly close to the target where most of the attack traffic will be arriving from a subset of the input links. That is a fairly safe assumption; even the biggest attacks do not involve more than a few thousand compromised machines, and there are many millions of machine on the Internet. It would be particularly hard for an attacker to ensure that the attack slaves are evenly distributed with respect to the target.

Another issue to examine is what fraction of the attack traffic originates from hosts served by the same ISP as the target. The smaller the ISP, the smaller that fraction will be, and even the largest of the top-tier ISPs will have a sizeable fraction of attacks originating from the outside. While an ISP can unilaterally deploy Pushback in its routers, unless agreements with its peering ISPs are made on how to honor pushback requests (an issue fraught with security and policy issues), said ISP will have to take advantage of pushback as best as it can. Now, in general, an ISP's network can be thought of as a cloud where clients attach (on edge routers) and which connects to other ISPs at peering points (private or public). An ISP's network can thus be viewed as a single virtual router, with multiple inputs and multiple outputs. If, in addition to output rate limiting, we were to implement input rate limiting, then the following variation of pushback could be considered: when an edge router detects an attack toward one of its attached customers, it tries to pushback determine what fractions of the attack traffic are coming through the border routers of the ISP. This could be done with some variation of ITRACE or marking by the border routers that would be caught and examined at the edge routers. Then, using (authenticated) tunnels to the border routers, the edge router would ask them to apply *input* rate limiting to the requested aggregate. If this is deemed undoable, input rate limiting on the border routers would still be useful in that it would effectively extend pushback by one more hop without the coöperation of the (upstream, belonging to a different ISP) router. Needless would still be useful for the part of the attacks originating from within the ISP's network.

6 Conclusions

We presented the implementation of a mechanism that treats Distributed Denial of Service attacks as a congestion-control problem, and acts by identifying and preferentially dropping traffic aggregates responsible for such congestion. The purpose of this work is twofold; show the practicality of such an approach, and explore ways to deploy it incrementally in an operational environment. We already know from simulations [MBF$^+$c] that Pushback is a promising way of combating DDoS attacks and flash crowds. There are some aspects that are easy to simulate, but real code running on real machines allows us to explore the details of a real system. We also needed to see how much memory and computing power is needed to actually run Pushback, in the hope of influencing commercial router designers toward implementing Pushback in their code. A promising hybrid solution, which we plan to investigate over the next few months, is to use features such as the Committed Access Rate[Cis98] in cisco routers to implement the rate-limiting, while sniffing traffic on both incoming and outgoing links of each router to detect congestion and dropped packets, even if the router itself cannot report those. Such experiments may allow rapid deployment of Pushback even in the absence of explicit support from router vendors.

Acknowledgments

The original idea for Pushback came from an informal DDoS research group consisting of Steven M. Bellovin, Matt Blaze, Bill Cheswick, Cory Cohen, Jon David, Jim Duncan, Jim Ellis, Paul Ferguson, John Ioannidis, Marcus Leech, Perry Metzger, Robert Stone, Vern Paxson, Ed Vielmetti, and Wietse Venema. A recent paper [MBF$^+$b, MBF$^+$a] presents the theoretical basis and detailed simulation results upon which this work has been based.

References

[BBC$^+$98] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An Architecture for Differentiated Services. RFC 2475, December 1998.

[BC00] Hal Burch and Bill Cheswick. Tracing Anonymous Packets to Their Approximate Source. In *Usenix LISA*, December 2000.

[Bel00] Steve M. Bellovin. ICMP Traceback Messages. Work in Progress, Internet Draft draft-bellovin-itrace-00.txt, March 2000.

[Cis98] Cisco Web Pages: Committed Access Rate. http://www.cisco.com/univercd/cc/td/doc/product/software/ios111 /cc111/car.htm, February 1998.

[CSZ92] David D. Clark, Scott Shenker, and Lixia Zhang. Supporting Real-Time Applications in an Integrated Services Packet Network Architecture and Mechanism. In *ACM SIGCOMM*, 1992.

[DFS01] Drew Dean, Matt Franklin, and A. Stubblefield. An algebraic approach to ip traceback,. In *Proceedings of NDSS '01*, February 2001.

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM SIGCOMM*, 1989.

[FBI$^+$] Sally Floyd, Steve Bellovin, John Ioannidis, Kireeti Kompella, Ratul Mahajan, and Vern Paxson. Pushback messages for controlling aggregates in the network. Internet Draft, *work in progress*.

[FBS] The FreeBSD Project. `http://www.freebsd.org`.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, Vol. 1(4):pp. 397–413, August 1993.

[FJ95] Sally Floyd and Van Jacobson. Link-sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking*, Vol. 3(4):pp. 365–386, August 1995.

[FS98] Paul Ferguson and Daniel Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2267, January 1998.

[MBF+a] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling high bandwidth aggregates in the network – extended version. `http://www.aciri.org/pushback/`.

[MBF+b] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling high bandwidth aggregates in the network. Submitted to Computer Communications Review.

[MBF+c] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling high bandwidth aggregates in the network. Submitted to ACM SIGCOMM 2001.

[MVS01] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring internet denial-of-service activity. In *10th Usenix Security Symposium*, August 2001.

[SDS00] Dan Schnackenberg, Kelly Djahandari, and Dan Sterne. Infrastructure for intrustion detection and response. In *Proceedings of the DARPA Information Survivability Conference and Exposition 2000*, March 2000. ftp://ftp.tislabs.com/pub/IDIP/DISCEX_IDR-Infrastructure.pdf.

[Sto00] Robert Stone. CenterTrack: An IP Overlay Network for Tracking DoS Floods. In *9th Usenix Security Symposium*, August 2000.

[SWKA00] Stefan Savage, David Wetherall, Anna Karlin, and Tom Anderson. Practical Network Support for IP Traceback. In *ACM SIGCOMM*, August 2000.

[Van97] Van C. Van. A defense against address spoofing using active networks. Batchelor's Thesis, MIT, 1997.