Introduction

The target domain of this project is defense against flooding-type distributed denial of service attacks (hereafter referred to as DDoS). The problem, as has become fairly common public knowledge over the past few years, is that these sorts of attacks are very difficult to defend against, both because of the semi-anarchic nature of the Internet, and because the number of attackers may grow to be very large. Several proposed solutions to this problem were topics for the course, but each of these is imperfect. Thus, I consider this area to be a challenging and interesting research domain.

The goals of my work were to implement four such DDoS defense schemes, and to evaluate their relative performance. These schemes were Hop Count Filtering (HCF) at routers, Pushback, a hybrid of HCF and Pushback, and an extension of this hybrid that uses IP traceback data from Probabilistic Packet Marking (PPM) to try to better choose routers for Pushback. These schemes will be discussed in further detail in later sections of this write-up.

Several assumptions were made in my project. These include correct knowledge of hop counts for an entire test network, and the ability to estimate bandwidth from the total non-dropped arrival rate of a traffic stream.

In completion of this project, I have implemented the four desired schemes. A performance evaluation is included in the results section of this paper, along with additional information.

Related Work

Some of the more interesting proposals (and hence those upon which my work is focused) to attempt to solve the DDoS problem that have been mentioned in class include ACC/Pushback [1, 2, 3, 5, 6] and Hop-count filtering [4]. Selective Pushback [7] is a proposed modification to Pushback.

Pushback is part of a scheme known as aggregate-based congestion control (ACC). ACC attempts to define aggregates (or categories of traffic with some common set of features), based on congestion signatures in the observed traffic at routers experiencing congestion on one or more output links. By identifying aggregates that account for a large amount of the (potentially attacking) traffic, and imposing additional rate-limiting on these aggregates prior to the normal output queue of the router, the local component of ACC attempts to eliminate congestion on output links by making it more difficult for the identified aggregates to overflow output queues. If rate-limiting at the edge router is not enough to affect a solution, pushback is invoked to proportionately rate-limit the aggregates at adjacent routers, eventually expanding the tree of routers involved until the congestion is reduced to a manageable level. There is great potential to penalize legitimate traffic that matches the congestion signature in this sort of solution, but Pushback is designed to avoid this collateral damage as much as possible. In practice,
aggregates in the currently-completed work in this area have been defined solely based on destination addresses in IP packets.

Hop-count filtering attempts to solve the problem in a different way. A database of hop-count information for all previously observed source addresses (by successful TCP three-way-handshakes) is built over time. When a large percentage of spoofed source addresses among sampled packets suggests a possible attack, hop-count filtering is invoked to eliminate packets with a presumed spoofed source address. The determination of spoofing, in this case, is made based on a comparison between the packet’s hop-count and the recorded hop-count for its source address. Hop counts are inferred based on time-to-live (TTL) values, using the knowledge that initial TTLs typically take on only a few specific values. The filters are instantiated either at lower levels in the protocol stack on the end host, or on incoming links at the host’s edge router. This helps to eliminate resources wasted on handling traffic that is likely undesired.

Selective Pushback, like the ACC schemes above, tries to identify and rate-limit aggregates responsible for congestion. Unlike the other ACC schemes, however, it tries to propagate rate-limiting sessions based on additional knowledge about their sources, rather than using a tree-like progression. The source of information used is IP traceback data obtained through PPM. The authors suggest that time-specific profiles of traceback data be recorded, and that decisions concerning rate-limit propagation during a period of high congestion might be made based on deviations from the expected traffic profile.

Methodology

My chosen architecture for the implementation of the previously mentioned schemes is a series of Router plugins for use with Simnet. These will be described in detail, but first I will describe the testing architecture and goals.

The authors of [5, 6] suggest that traffic be categorized as bad, poor, and good. Bad traffic is attack traffic. Poor traffic is legitimate traffic that is, unfortunately, destined for the same host as some flow(s) of bad traffic. All other traffic is good. Our goal is to attempt to protect good and poor traffic from high loss rates caused by congestion on links saturated by bad traffic. Thus, for the results described in the next section, the statistics of interest are the ratio of traffic sent to traffic received for each of these categories.

The topology used in my tests is shown in Figure 1. This topology is meant to model the network of an Internet Service Provider (ISP). At the top, hosts representing other networks inject traffic of the type for which they are named. Traffic then flows through a well-connected mesh of routers, until it is concentrated onto low-bandwidth links at the network edge. For this reason, the links between router 19 and router 21, between router 20 and router 22, and between routers 21 and 22 and their adjacent hosts, have a queue size of 10, as opposed to the default queue size of 1000 for other links in the topology.
Figure 1: The test topology used, contained in networks/PushbackTestNetwork.odf.

The scripts and Application (TestSender and TestClient) plugins used to test my Router plugins inject packets into the network at a constant rate. Poor and good hosts (at the top) each send 100 packets per second; two poor hosts send to the victim host (bottom left), while two good hosts send to each of the neighbor hosts (remaining hosts at bottom). Bad hosts each send 1000 packets per second to the victim host; these packets have a source address chosen at random from among the valid addresses for the test network. These transmission rates will need to be varied to perform these tests depending on the platform used, because there is no notion of bandwidth in the current version of Simnet; the relevant data item is the service rate of each output queue, dictated by a combination of processing power, threading support, and implementation efficiency. Two sets of tests were performed: one in which the attack was continued uninterrupted for 1 minute, and one in which the attack was stopped after 1/2 minute, leaving the network to legitimate traffic for the remainder of the test, in order to determine if the defense schemes accurately determine when they are no longer needed.

For a simple Hop Count Filtering Router (HCFRouter), I wrote code to generate a correct mapping between hop counts and node addresses. The router plugin observes a sample of ten percent of the incoming traffic, and records the observed rate of spoofed source addresses over a ten second window. If the spoof-rate rises above a threshold of twenty percent, the plugin begins to actively monitor all incoming traffic, rather than just a sample, and drops any packet with a source address that is identified as spoofed. In my implementation, a priori knowledge of correct hop counts and a lack of meaningful address clustering make it possible to avoid all false positives. However, false negatives
are still a concern, because there is possibility that the distance between the actual source node and the router happens to be the same as the distance between the packet’s nominal source and the router. In [4] it was suggested that hop counts in the internet are roughly normally distributed. I would expect my implementation to be fairly effective in filtering spoofed packets, but unable to protect poor traffic or good traffic destined to neighbors of the attack target from an unusually high loss rate due to upstream congestion, since this mechanism only takes place at edge routers.

For ACC/Pushback, I implemented a PushbackRouter plugin that delivers reports of arrived and dropped packets, and performs rate-limiting in the appropriate location, using the mechanisms provided by an Application plugin, PushbackD. PushbackD implements a variant of the local ACC algorithms provided in [5], as well as Pushback. It records all arrived and dropped packets, and periodically (every two seconds) evaluates the need for new local rate-limiting sessions. Rate-limits are adjusted whenever the rate of dropped packets is above a threshold of ten percent, with the goal of returning to an acceptable threshold of five percent; this process is lower-bounded by the requirement that the limit for an aggregate must not be lower than the highest arrival rate of a non-limited aggregate. Every five seconds, aggregates that have shown a sustained high drop rate even after rate-limiting are “pushed back.” That is, the limit is divided according to arrival rates among each of the routers with incoming links to the congested router. This is done through the mechanism of Pushback REQUEST (sent the first time) and REFRESH (sent after the first time) messages. Upstream routers who receive these messages adhere to the rate limit given, and provide STATUS messages to their parents in the Pushback tree. The timing of STATUS messages is such that the rate-limiting session’s initiator (at the root of the tree) can receive an update every two seconds; this requires upstream messages to be sent faster (by one round-trip-time for each hop). Rate-limits are removed at upstream routers after thirty seconds (fifteen seconds in tests, in order to speed things up) pass without receiving a REFRESH message; the initiator only sends REFRESH messages when a high drop rate is observed locally, or through the STATUS mechanism.

My implementation has several things worth mentioning. First, of the several types of propagation types specified by [2], only HI_DROP_PROP is supported; that is, upstream routers only propagate rate-limiting sessions if they observe a high drop rate. Second, only compact STATUS messages are used, which estimate the total arrival rate for the aggregate in a sub-tree of the Pushback tree. Third, rate limits are never raised; they are only lowered if drop rates suggest this is necessary. The reasoning behind this last decision is that a rate limit that is too low only hurts traffic that exceeds the limit, while a limit that fluctuates too frequently may allow an attacker to exploit this interval and inflict congestion on other traffic. A fourth item of interest is that my rate limiting is enforced by a mechanism called preferential dropping, which drops each packet belonging to a limited aggregate with probability $1 - \left(\frac{\text{rate}_{\text{limit}}}{\text{rate}_{\text{arrival}}}\right)$. I would expect Pushback as implemented to do a fair job of protecting good traffic, and to do somewhat better than HCF for poor traffic.
In addition to the Pushback implementation, classes for each of the Pushback message types were implemented, including CANCEL in addition to those described above. CANCEL messages are not sent by my implementation, as I regard their use as a policy decision.

My implementation of Pushback with HCF filtering is fairly straightforward. It builds a hop count to IP address mapping in the same manner as the “vanilla” HCFRouter. However, rather than employ the same thresholding schemes, HCFPushbackRouter depends on its HCFPushbackD to identify aggregates that should be rate limited, and their limits, in the identical manner to PushbackD. The difference is that, for aggregates that are being limited, all spoofed packets are dropped before the rate-limiting is used. This should provide a significant benefit to poor traffic, by allowing it to consume more of the rate limit.

My other hybrid implementation, SelectiveHCFPushbackRouter and SelectiveHCFPushbackD, attempts to make better decisions about where to propagate rate limits, similar to the approaches described in [7]. My implementation, in this case, works as follows. PPM traces are recorded only for routers that I know to be on the edge of the network based on the TTL of messages that arrive with their ID. This uses the information in the hop count table to simulate the effect that a distance field (a normal component of the edge sampling PPM scheme, as opposed to the node sampling scheme present in Simnet currently) combined with the knowledge of my pushback network’s boundaries would have. Basically, the Pushback initiating router sends its requests directly to these edge routers, and these routers propagate the session as normal. The rate limit is divided not based on the initiating router’s incoming link arrival rates, but rather according to an arrival rate for each of these routers, as estimated from the percentage of marked packets and hop count information of each. Some additional handling is also required for the initiator to interpret STATUS messages from these edge routers. I expect the performance of this hybrid to exhibit a slight improvement over the previously described hybrid.

No major extensions to Simnet were required for my project. Router plugins overrode the run() and out() methods as necessary in order to add hooks for the Pushback daemons. RoundRobin was subclassed to create a LinkReportingRoundRobin link processor that would allow an Application to determine the incoming link from which a packet was dequeued.

**Results and Analysis**

The results as measured have largely validated the hypotheses described in the previous section. Before going into details, it is appropriate to restate assumptions. As stated earlier, assumptions were made such that full hop count knowledge was available, that bandwidths could be estimated from non-dropped arrivals, and that it was acceptable to simulate a PPM scheme with distance marking using hop counts and some knowledge of the topology structure.
Table 1: the result of the first round of tests, with full-length (60 second) attacks, as described in the section above.

<table>
<thead>
<tr>
<th></th>
<th>good sent</th>
<th>good received</th>
<th>poor sent</th>
<th>poor received</th>
<th>bad sent</th>
<th>bad received</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>36065</td>
<td>34049</td>
<td>12018</td>
<td>10167</td>
<td>240294</td>
<td>158534</td>
</tr>
<tr>
<td>HCF</td>
<td>36032</td>
<td>28999</td>
<td>12007</td>
<td>7791</td>
<td>239518</td>
<td>30957</td>
</tr>
<tr>
<td>Pushback</td>
<td>36038</td>
<td>34233</td>
<td>12007</td>
<td>8451</td>
<td>240078</td>
<td>138360</td>
</tr>
<tr>
<td>HCFPushback</td>
<td>36027</td>
<td>35572</td>
<td>12004</td>
<td>9397</td>
<td>240084</td>
<td>21657</td>
</tr>
<tr>
<td>Selective</td>
<td>36032</td>
<td>35697</td>
<td>12006</td>
<td>11635</td>
<td>240070</td>
<td>15770</td>
</tr>
</tbody>
</table>

The results of the first experiment are shown in Table 1 and Figure 2. The baseline result, with normal Router plugins, appears anomalous, since it does better than each of the defense schemes. However, it seems likely that this is caused by the increased service rate of the output queues that can be obtained with less per-packet processing. The remainder of the results appear to match expectations.

Hop count filtering does a good job of preventing the bad packets from arriving, but is unable to do an acceptable job of protecting the poor and good traffic. This result is to be expected, as HCF takes place only at the end router, and cannot prevent congestion upstream. Pushback doesn’t block nearly as much of the bad traffic, but does a significantly better job with good traffic, and shows some improvement with poor traffic. This, again falls into the category of expected results; pushback doesn’t have the ability to target bad traffic by its spoofed nature, but does protect good traffic by limiting the congestion that can be caused by rate-limited aggregates. Also, by propagating the rate limits upstream, the sources of the attack can be better isolated, allowing improvements for poor traffic.

![Experiment 1 - Full-length Attacks](image)

Figure 2: The results of the first experiment, displayed as percentage received for each traffic type.
The HCF-Pushback hybrid combines the two congestion-minimizing approaches to help good traffic significantly, and also manages to help poor traffic as well. Part of the benefit to poor traffic is likely due to the fact that, thanks to HCF on the output link before the rate limiter, poor traffic (which is not spoofed) is able to get a larger share of the rate limit.

The Selective HCF-Pushback hybrid shows an even more significant improvement. By propagating ACC straight to the entry point of the bad traffic onto the test network, it is able to protect poor traffic to an unexpectedly high degree. Good traffic is also nearly perfectly protected.

Table 2: the results of the second round of tests, with attacks that only last for the first 30 seconds, as described in the previous section.

<table>
<thead>
<tr>
<th></th>
<th>Good sent</th>
<th>good received</th>
<th>poor sent</th>
<th>poor received</th>
<th>bad sent</th>
<th>bad received</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>36030</td>
<td>35145</td>
<td>12007</td>
<td>11245</td>
<td>120042</td>
<td>80770</td>
</tr>
<tr>
<td>HCF</td>
<td>36035</td>
<td>33419</td>
<td>12007</td>
<td>10307</td>
<td>119847</td>
<td>18788</td>
</tr>
<tr>
<td>Pushback</td>
<td>36043</td>
<td>34996</td>
<td>12005</td>
<td>9164</td>
<td>120026</td>
<td>98371</td>
</tr>
<tr>
<td>HCFPushback</td>
<td>36030</td>
<td>35575</td>
<td>12005</td>
<td>10834</td>
<td>120044</td>
<td>18141</td>
</tr>
<tr>
<td>Selective</td>
<td>36019</td>
<td>35754</td>
<td>12005</td>
<td>11683</td>
<td>120042</td>
<td>11766</td>
</tr>
</tbody>
</table>

The second experiment also shows unexpectedly good results for the baseline, likely for the same reasons. The real concern here is showing how well the different schemes adapt to the end of an attack. Arrival ratios should be higher for each group, because congestion was minimal during the second half of the experiment time. However, some schemes did a better job than others of avoiding residual effects.

HCF only affects spoofed traffic. For this reason, it had no residual effects on good or poor traffic. HCF saw a large spike in its arrival ratio for good and poor traffic over the first experiment because it wasn’t doing anything to mitigate upstream congestion; thus, the lack of such congestion proved to be especially helpful. Pushback, on the other hand, didn’t do better by as much when compared to experiment one, likely because it took approximately fifteen seconds for the rate limits to expire after the attack ceased, causing some collateral damage to the poor traffic.

Pushback with HCF showed the expected behavior for good traffic, but improved for poor traffic. I’m uncertain as to why, however, it improved more than ordinary pushback. HCF-Pushback’s rate limits last just as long as Pushback’s, and should consequently have a similar residual effect. From an examination of the experiment logs, it appears likely that some effect caused HCF-Pushback to propagate further and impose additional rate limits on the poor traffic streams during experiment one, skewing the comparison.

Selective Pushback with HCF shows its benefits here as well. Because the rate limits happen so close the attacking hosts, poor traffic doesn’t suffer very greatly at all, despite the fact that the rate limits last for around fifteen seconds after the attack ends.
Experiment 2 - Half-length Attacks

Figure 3: The results of the second experiment, displayed as percentage received for each traffic type.

The implementation of this project faced significant hurdles in adapting the algorithms described in the referenced sources to a system with no concept of bandwidth as an explicit limit. Working around this problem required a logical shift in measurement ideas. Other difficulties faced include the variability of service rates at the output queue across hardware and software platforms, making it difficult to define testing parameters. However, it seems that a successful project has resulted, regardless.

References


**Experience with Simnet**

I will provide this information through email afterwards. Time constraints prevent me from writing an adequate response before completing the project.