2002-09

A method for mitigating denial of service attacks on differentiated services networks

Braun, Matthew J.

Monterey, California: Naval Postgraduate School, 2002.

http://hdl.handle.net/10945/9798

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A METHOD FOR MITIGATING DENIAL OF SERVICE ATTACKS ON DIFFERENTIATED SERVICES NETWORKS

by

Matthew J. Braun

September 2002

Thesis Advisor: Geoffrey Xie

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE *Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
SEP 2002

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE: A Method For Mitigating Denial Of Service Attacks On Differentiated Services Networks

5. FUNDING NUMBERS

6. AUTHOR(S) Matthew Braun

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

This thesis presents a method for countering Denial of Service (DoS) attacks in networks that provide Quality of Service (QoS) guarantees using Differentiated Service (DiffServ). This approach uses feedback from the DiffServ provider to initiate packet signing at the source. The signature allows the DiffServ provider to distinguish valid packets from malicious packets. This mechanism can also be used to provide key management for other digital signature methods, such as the Internet Protocol Authentication Header (IP AH). However, unlike other methods, our solution requires no encryption or cryptographic processing on a per-packet basis. Instead, it utilizes the sender's ability to alter its packet signatures faster than the attacker can duplicate the changes. This method also avoids the fragmentation and decreased throughput associated with increased packet size of IP AH through use of existing fields in the IP header. This method results in a significant reduction in valid packets that are dropped during a DoS attack. Thus, a DiffServ provider would be able to maintain QoS guarantees during an attack without incurring the overhead associated with cryptographic signatures. A C++ implementation of this DoS countermeasure for the ns2 network simulator and the experimental simulation scripts are included as appendices.

14. SUBJECT TERMS
Differentiated Service, DiffServ, Denial of Service, DOS, Quality of Service, QOS, Networks, NS2

15. NUMBER OF PAGES
101

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UL

THIS PAGE INTENTIONALLY LEFT BLANK

A METHOD FOR MITIGATING DENIAL OF SERVICE ATTACKS ON DIFFERENTIATED SERVICES NETWORKS

Matthew J. Braun
Lieutenant, United States Navy
B.S., University of Illinois at Urbana-Champaign, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2002

Author: Matthew Braun

Approved by: Geoffrey Xie
Thesis Advisor

Richard Scott Cotè
Second Reader

Christopher Eagle, Chairman
Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis presents a method for countering Denial of Service (DoS) attacks in networks that provide Quality of Service (QoS) guarantees using Differentiated Service (DiffServ). This approach uses feedback from the DiffServ provider to initiate packet signing at the source. The signature allows the DiffServ provider to distinguish valid packets from malicious packets. This mechanism can also be used to provide key management for other digital signature methods, such as the Internet Protocol Authentication Header (IP AH). However, unlike other methods, our solution requires no encryption or cryptographic processing on a per-packet basis. Instead, it utilizes the sender's ability to alter its packet signatures faster than the attacker can duplicate the changes. This method also avoids the fragmentation and decreased throughput associated with increased packet size of IP AH through use of existing fields in the IP header. This method results in a significant reduction in valid packets that are dropped during a DoS attack. Thus, a DiffServ provider would be able to maintain QoS guarantees during an attack without incurring the overhead associated with cryptographic signatures. A C++ implementation of this DoS countermeasure for the ns2 network simulator and the experimental simulation scripts are included as appendices.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The author wishes to thank his wife Krista for her love and support during the research and preparation of this thesis.

The author wishes to acknowledge Prof. Geoffrey Xie for his advice, support, and instruction, without which this thesis would not have been possible.

This research was supported in part by DARPA under the Next Generation Internet Program (AO#417) and by the National Science Foundation under grant No. ANI-0114014.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION.

A. BACKGROUND

Future military networks such as the Navy's FORCENet concept will carry traffic between many dissimilar groups of users simultaneously. Traffic may be operational or administrative, and of varying precedence. Military networks must be able to determine which traffic should receive higher priority. For example, traffic between forces in combat must be given priority over routine administrative traffic. The current model of Internet routing provides only Best Effort service. No guarantees are made with respect to how, when, or if traffic will reach its intended destination. Traffic that requires Quality of Service (QoS) guarantees - priority over other traffic or specific bounds with respect to transmission quality - is at a disadvantage in a best effort environment. Differentiated Service (DiffServ) is a method of providing QoS guarantees to network traffic by aggregating similar traffic and giving priority to specific aggregates.

A Denial of Service (DoS) occurs when users are prevented from utilizing a service provided by a system. The most widespread method of creating a DoS is by artificial exhaustion of a resource, such as bandwidth, processor cycles, or memory. A Distributed Denial of Service (DDoS) attack is one in which an attacker uses the combined power of many hosts to exhaust the resources of a server. New types of DoS attacks will accompany implementation of the DiffServ model. The separation of traffic into aggregates will make it easier for an attacker to target a specific subset of traffic flowing in the network. DiffServ also offers new possibilities for the prevention of DoS attacks, since a DiffServ provider will have more information about its clients than is currently tracked using today's implementation of the protocols found on the Internet.

B. PURPOSE

The primary goal of this thesis is to develop and test a mechanism for preventing DoS attacks in DiffServ networks. Secondary goals necessary to accomplish this include determination of the topologies likely to be used in future DiffServ networks, examination of the specific vulnerabilities of these topologies to DoS attack, creation of an analytic model that can be used to calculate the effectiveness of a DoS countermeasure, and implementation of the DoS countermeasure as an extension of an existing network simulator.

C. SCOPE

The scope of this thesis is limited to (a) a review of the Differentiated Services standard, (b) determination of the constraints DoS networks will impose on new DoS attacks, (c) development of a feedback-based DoS countermeasure for use by DiffServ clients and providers, and (d) implementation and testing of the countermeasure in the ns2 network simulator.

D. MAJOR CONTRIBUTIONS

This thesis explores the possible changes in the manner in which DoS attacks are conducted that will accompany widespread implementation of DiffServ in the next-generation Internet. A technique is presented for countering DoS attacks against DiffServ networks. This technique can also be used in certain cases to detect illegitimate use of premium services without payment. An implementation of the technique in a common open-source network simulator is provided. The theoretical performance of this technique is derived mathematically and compared to the results obtained by simulation. The communication method used in this technique may be useful in other applications as well.

E. RELATED WORK

A summary of this research was submitted to the 10th International Conference on Telecommunications Systems, Modeling, and Analysis (ICTSM 10) as a paper entitled *A Feedback Mechanism for Mitigating Denial of Service Attacks against Differentiated Services Clients* [BRAUN02]. This thesis presents the material covered in that paper and expands upon it.

F. ORGANIZATION

The remainder of this thesis is organized as follows:

- Chapter II is an overview of network QoS in general and DiffServ in particular.
- Chapter III is an examination of attacks used in the existing Internet and the methods proposed to defeat them. The effectiveness of these attacks and methods

in DiffServ networks is discussed. Differences in DoS attacks and the scenarios in which they would be possible and effective against a DiffServ network are discussed.

- Conceptual details of the DoS countermeasure developed by this thesis are given in Chapter IV. This chapter also includes a derivation of the theoretical effectiveness of the countermeasure
- Chapter V describes how the proposed DoS countermeasure was implemented. It includes a description of the design and architecture of the ns2 network simulator. Specific modifications to the simulator are discussed in detail.
- Chapter VI sets for the simulated topology, experimental design, and simulation results.
- Chapter VI lists the conclusions and future work for the thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

II. NETWORK QUALITY OF SERVICE

A. OVERVIEW

Providing Quality of Service guarantees is one of the next steps in the evolution of Internet routing. Networks that provide QoS guarantees allocate resources (queue space, bandwidth, etc.) to traffic flows differently based on the source and type of each flow. The performance of applications such as streaming audio and video, Voice over IP (VoIP), and video teleconferencing can be degraded by network congestion or losses. Implementation of QoS guarantees can eliminate degradation and ensure fairness for all flows. Current and proposed types of service are discussed below.

B. BEST EFFORT SERVICE

The current model of Internet routing provides only best effort service – no guarantees are made with respect to jitter (inter-packet arrival time), latency (delay between transmission and receipt), error rate, or loss rate (number of dropped packets). Best Effort service does not prioritize one type of traffic over another, even in cases where it could be done easily and at low cost. All traffic is treated equally, which puts traffic that requires limitations on the aforementioned qualities at a disadvantage. Time-sensitive traffic can be delayed or lost due to congestion caused by applications that do not have any specific latency requirements.

Best Effort service does have certain advantages. It does not require routers to set up end-to-end connections or maintain state information on a per-flow basis. Best Effort routers also do not need to allocate or manage multiple queues for multiple traffic flows.

C. INTEGRATED SERVICE

One proposed alternative to Best Effort service is to monitor each end-to-end connection, hereinafter referred to as a flow, and selectively determine the order and direction in which packets are transmitted based on the QoS guaranteed to each flow. This provides a much finer granularity of service, since it allows prioritization at the flow level. However, this method is not feasible for use in the Internet because it is not scalable to large networks. It requires end-to-end connection setup for each flow, which in turn requires every router along the path of the flow to maintain state information about the flow. In the network core, where the paths of many flows overlap and traffic volume is very large, routers do not have the queue space or processing resources that would be required to monitor every flow passing through them.

D. DIFFERENTIATED SERVICE

Differentiated Service is a method of providing QoS to traffic flows without having to the maintain flow state information at every router. Traffic classification is distributed to the edges of the network where volume is lighter. Edge routers police traffic entering a network based on the Service Level Agreement (SLA) between the source and the domain operator. Traffic is classified and conditioned to conform to one of a fixed number of specific behavior aggregates, which are pre-defined throughout the DiffServ domain. Per-hop behavior (PHB) is defined as “a description of the externally observable forwarding treatment applied at a differentiated services-compliant node to a behavior aggregate”[RFC2474]. The treatment that traffic assigned to an aggregate will receive at core routers is determined by the PHB associated with that aggregate. This is known as core-stateless routing. Core routers do not track the state of individual flows. They are only responsible for forwarding based on the classification assigned to each packet when it entered the network.

1. Architecture

DiffServ has not yet been implemented on a large scale in the Internet. Implementation will not happen all at once, but will occur in individual domains as service providers upgrade equipment and software in their networks. A DiffServ domain is defined as

> “…a contiguous portion of the Internet over which a consistent set of differentiated services policies are administered in a coordinated fashion….”[RFC2474]

The DiffServ code-point is defined as the first six bits of the eight-bit Type of Service (ToS) field in a packet’s IP header. Each behavior aggregate is identified by a single DiffServ code-point. When a packet reaches the ingress router of a DiffServ domain, the ingress router will mark it by changing the value of the DiffServ code-point to the code-point associated with the correct aggregate for that packet. Packets from unrecognized sources, i.e. those that do not have an SLA with the domain, are assigned a default code-point. Core routers treat packets with the default code-point as Best Effort traffic, thus ensuring backwards compatibility for traffic arriving from non-DiffServ domains.

It is important to note that all routers in the DiffServ domain that are linked to other domains must be configured as ingress routers. This provides access control to the QoS guarantees provided by the DiffServ domain. Within the core of the network, packets are forwarded according to the per-hop behavior associated with the DiffServ code-point [RFC2474]. Traffic entering a core router without first being marked by an ingress router would be forwarded based on the code-point it arrived in the DiffServ domain with. This would allow service theft - a situation in which some traffic receives preferred treatment that it is not entitled to receive. Service theft could be accomplished by setting the traffic's code-point to one that receives premium treatment by the DiffServ core routers.

2. Possible DiffServ Topologies

The proximity of the client and receiver to the DiffServ network affects the path that DiffServ traffic will follow. This path in turn affects the guarantees the provider can make to clients. There are three types of connection paths possible, as shown in Figure 2.1.

a. End-to-End Single Provider Path

In this topology, clients and receivers are directly linked to a single DiffServ domain. Traffic receives QoS guarantees over its entire path. These guarantees are offered and ensured by a single DiffServ provider. In Figure 2.1, traffic from A to B would follow this type of path.

b. End-to-End Multiple Provider Path

In this topology, clients and receivers are connected to different DiffServ domains. Traffic flows on a path through different domains, but still receives QoS guarantees over the entire path. All individual domains along the path must agree to provide QoS guarantees to traffic arriving from upstream DiffServ domains. In Figure 2.1, traffic from B to D follows this type of path.

c. Partial Best-Effort Path

In this topology, the traffic from client to receiver transits a non-DiffServ-capable domain at some point. QoS is provided only during the times when the traffic is in a DiffServ domain. In Figure 2.1, traffic from C to D and from A to C follows this type of path.

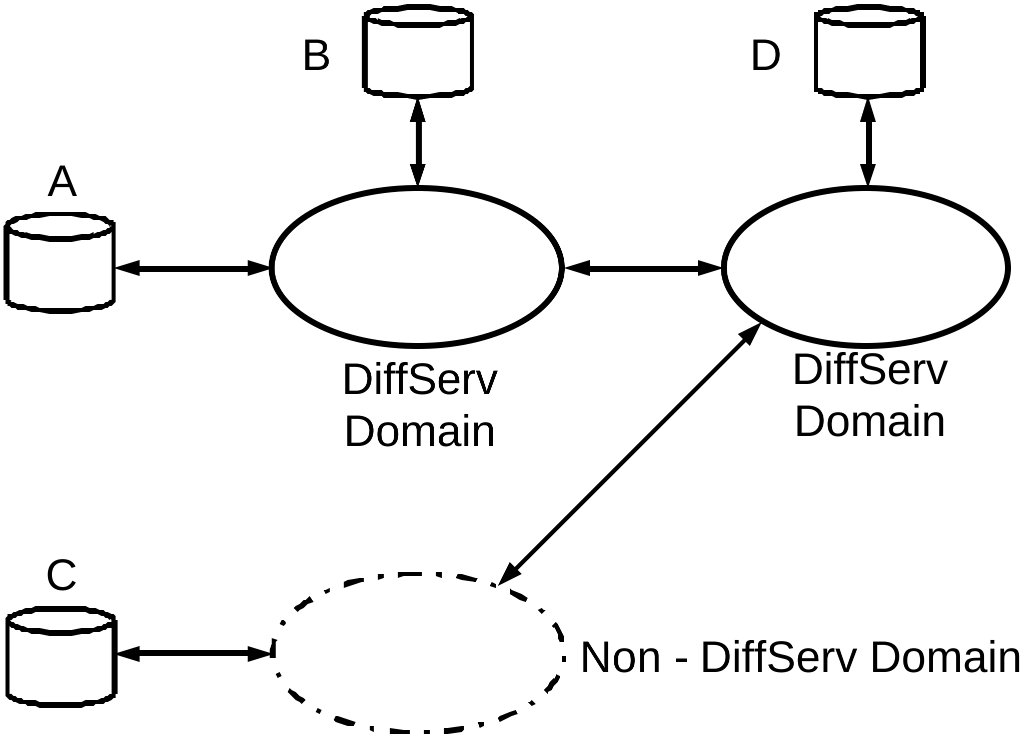

Figure 2.1. Possible Network Traffic Paths.

III. DENIAL OF SERVICE ATTACKS

A. OVERVIEW

The goal of Denial of Service attacks is to prevent users from utilizing a service that a system is providing. DoS attacks may target individual flows and groups of flows or they may be indiscriminate. They may require the compromise of network devices, i.e. hosts, servers, and routers. These devices may not have to be along the path of a targeted flow in order for an attack to be effective. General types of interference, tactics that support or enable attacks, methods of interfering with or denying services, and the feasibility of these methods are discussed below.

B. TYPES OF INTERFERENCE

Denial of Service can be accomplished by disabling network hardware or software, by directly manipulating user traffic, or through misuse or abuse of legitimate protocols that interferes with the flow of user traffic. Network interference, traffic interference, or both can be used to create DoS attacks.

1. Network Interference

Network interference is the result of rendering network hardware or software inoperative. This disruption stops the normal function of a part of the network. Exploitation of flaws in network software that cause it to stop operating properly or halt is an example of software interference. Hardware interference can be accomplished by

- cutting a network link
- removing power from a network device
- jamming frequencies used by wireless access points.

Although network interference may have a disproportionate affect on certain traffic sources or destinations due to their proximity to the disruption, it is usually indiscriminate since it affects all traffic attempting to transit the damaged portion of the network. An attacker would not need to compromise any network devices to create this kind of interference.

2. Traffic Interference

Traffic interference consists of any improper delay, manipulation, or dropping of packets. An attacker can interfere with traffic either directly or indirectly.

a. Direct Interference

In order to directly interfere with network traffic, the attacker must compromise a network device and alter the way the device performs. This type of interference may be targeted or indiscriminate. For example, if a router is compromised and instructed to drop every third packet received, the attack would be indiscriminate. A targeted attack would consist of interfering with traffic originating from or destined to a specific host or hosts. If the attack is targeted, the compromised device must be along the path of the target flow(s).

b. Indirect Interference

Unlike network or direct traffic interference, indirect traffic interference does not disrupt operation of network devices and protocols along the path of the target flows. The attacker attempts to exhaust some resource, such as link bandwidth or router memory, by overwhelming it with apparently valid traffic. Once the resource reaches 100% utilization, any additional traffic attempting to use it must be dropped. The percentage of valid traffic that will be dropped is inversely proportional to the ratio of the demand for the resource to its capacity. For example, if the attacker can raise the demand for a link to 10 times its capacity, then only 1 in 10 packets arriving at the link will be transmitted. This method is indirect because harm to valid traffic is the result of processing invalid traffic, not direct manipulation.

C. ENABLING TACTICS

This section discusses two tactics that do not constitute DoS attacks themselves, but which are widely used in the conduct of DoS attacks.

1. IP Header Manipulation

IP header manipulation is an enabling tool for the majority of indirect DoS attacks. As the name suggests, it consists of altering fields in the IP headers of packets. Under normal conditions, a host will place its IP address in the IP header Source Address field. For most methods of attack, the IP Source Address field of outgoing packets is changed. This is commonly known as address spoofing.

Source address spoofing serves two purposes in DoS attacks. First, it makes it impossible for a receiver to determine the true source of an attacker's packets. Second, any replies to the source of packets with spoofed IP addresses will be sent to the spoofed address in the header instead.

2. Compromised Hosts

Network or Internet servers must be able to handle connections from hundreds or thousands of clients simultaneously. Therefore, the bandwidth of inbound links to servers must be several orders of magnitude larger than the outbound bandwidth of individual clients. This makes it very unlikely that an attacker will be able to conduct a bandwidth consumption attack by using a single host. Most indirect interference attacks require that production of malicious traffic be spread among hundreds of individual hosts; hence the term distributed DoS.

An attacker can obtain a DDoS flood source by breaking into a host and installing a DDoS attack tool. Once the tool has been installed, the host will respond to commands sent from an attack manager under the attacker's control. The attack manager, which is also a compromised host, maintains a list of the compromised flood sources. To initiate and control an attack, the attacker sends commands to the manager, which in turn sends commands to the individual hosts. These commands include the type of attack to be conducted, start and stop times, and the target of the attack. Use of this three-tiered system makes it difficult or impossible to trace an attack back to the perpetrator.

D. ATTACK PATH

DoS attack traffic may flow directly from source to target, or it may be sent to an intermediate host. The latter indirect path is a characteristic of reflected attacks (RDoS or RDDoS). When sending packets to an intermediate host, the attacker spoofs the target's IP address in source address field of the attack packets. The third-party host becomes an unwitting participant in the attack by sending a reply to what it thinks is the source of the packets, but which is actually the attack target. Using a reflector in this manner further obfuscates the true source of the attack.

E. ATTACK METHODS

Methods of creating a denial of service vary based on the equipment and protocols being used by the service provider and the resources available to the attacker. Attack methods based on network interference or direct packet manipulation are both possible and highly effective. However, these types of interference must generally be countered through implementation of physical security or access control mechanisms, which are beyond the scope of this thesis. This section focuses on various known methods of indirectly attacking network traffic.

1. TCP Flooding

The TCP handshake consists of a series of request and acknowledgement messages used to set up a TCP connection between a client and server. A client initiates a TCP connection request by sending a SYN segment to the server. When the server receives a SYN segment, it attempts to create a new connection by allocating resources, replying with a SYN/ACK segment, and waiting for a reply from the SYN originator. If no reply is received, the connection will eventually timeout and the resources allocated to the connection will be released.

To exploit this protocol, the attacker floods the target with SYN segments with invalid and unreachable source addresses. A host will normally send a RESET segment to the source of an unexpected TCP segment. The RESET segment allows the original source to release resources reserved for the connection. However, since the requesting address is unreachable, no RESET reply to the target’s SYN/ACK segments will ever be received. If the attacker can send enough requests to use up all available resources before incomplete connections begin to timeout, the target will not be able to accept any further SYN segments, resulting in a DoS to legitimate clients. Some variations of this attack use randomly spoofed source addresses to get past filters on the target system that are set up to stop packets originating from a single source. This type of attack follows a direct path to the target.

Various other malformed TCP segments can be flooded at a host to elicit RESET responses. This can be used in an attempt to consume the available bandwidth between a target and its connection to the Internet, or as a reflected attack against another host.

The effect of TCP flooding is generally indirect interference, since the processing of invalid requests prevents service to valid requests. However, it can cause network interference if exhausting the server's resources causes it to stop functioning.

2. Chargen Attack

The UDP chargen service responds to a UDP echo request by sending a string of characters to the source of the request [RFC864]. An attacker can exploit this by spoofing the target's IP address in a UDP echo request. This request is sent to the chargen port on an intermediate system. Receipt of the request will create a feedback loop between the intermediate and target systems, preventing either one from responding to legitimate requests. This indirect-path attack results in indirect interference.

3. ICMP Flooding (Smurf Attack)

When a system receives an ICMP echo request, it will respond with an ICMP echo reply. However, if the request is addressed to a network broadcast address, it will be forwarded to every host in that network, and each host will send an echo reply. To take advantage of this, an attacker sends an ICMP echo request, with the target's IP address forged as the source address, to a network broadcast address. This intermediate victim will broadcast the echo request to all hosts in the network, and they will all send echo replies back to the (forged) source of the request. Since one ICMP packet sent to the broadcast address will be multiplied into hundreds or thousands of packets sent to the target, flooding the intermediate broadcast address with echo requests can rapidly consume the bandwidth of the target's upstream link.

F. PROPOSED DOS COUNTERMEASURES

1. Filtering

There are two main types of filtering, ingress and egress, that can be effective against DoS attacks. Ingress filtering refers to filters applied to traffic as it enters the Internet [RFC2827]. It is used to prevent IP address spoofing. Only packets with routable source addresses that lie within the address range of a site are allowed to pass though the filter. This type of filter does not completely prevent forged source addresses. An attacker can still use forged source addresses within the originating domain's assigned address range. However, filters of this type make it easier to find the true source of packets with forged header information.

Ingress filtering is one of the more effective means of preventing DoS attacks, since it stops the address spoofing most attack rely upon. However, there are several limitations to this approach. For ingress filters to be truly effective, they must be implemented by a large majority of Internet Service Providers (ISP's). This will be hard to achieve, since it requires service providers to implement controls that do not directly benefit them. Additionally, ingress filtering does not affect packets with spoofed IP source addresses in the same range as those assigned to the network implementing the filter.

Egress filtering is performed at the destination. Filters are set up to drop malformed or unauthorized packets as they are received. Egress filters are primarily used to prevent network interference attacks designed to exploit flaws in applications. They are ineffective at preventing indirect DoS attacks, since these attacks generally use valid requests that cannot be blocked while maintaining service to legitimate users. A victim could use an egress filter to block all traffic from the domains that spoofed packets originate from, but this would still cause a DoS to legitimate users in that network. Furthermore, dropping packets as they are received at the destination does not relieve congestion on the link between the target and its upstream router(s).

2. IP Traceback

IP Traceback is a method of determining the true source of a packet with a forged source address. When it is implemented, Internet routers would randomly mark packets with their own IP address as the packet passes through them. After receiving a sufficient number of marked packets, the victim would be able to determine the full or at least partial path taken by malicious packets. Once the true source IP address was known, the victim could work directly with the ISP of the source to stop the attack. The advantages of this defense are low processing overhead required for implementation, backwards compatibility with existing protocols (the marking can be placed in a field of the IP header), and the ability to determine the true source of malicious packets without active cooperation from the administrator of every router through which the packets pass [LEE01, SAVAGE00].

There are several drawbacks to this countermeasure. First, it does not provide a way to distinguish between valid traffic and malicious traffic, so valid traffic may be penalized. Second, some minimum number of malicious packets from a source must be marked in order to trace the flow from that source. A resourceful attacker could take advantage of this by limiting the number of packets sent from any individual compromised host. Finally, IP Traceback does not provide any relief to the victim during an attack. Until the victim can stop the flood of the malicious packets, the attack will be effective. Stopping the flow requires cooperation from the ISP at each source of flooding traffic.

3. Router Throttling

Router throttling is a method of distributed flow control. It is designed to reduce traffic destined for a specific host at a point several routers upstream of the aggregation point for receipt by the host [YAU01]. When a server is overloaded with traffic, it calculates a maximum permissible flow rate and sends this value to participating upstream routers. The upstream routers limit traffic destined to the overloaded server based on the specified flow rate. In some cases, upstream routers may pass the flow rate limits to other routers further upstream. This countermeasure shifts the processing requirements for large amounts of traffic from the target server to routers with greater bandwidth and filtering capability.

Router throttling relies on a large measure of cooperation from other systems. It also requires the victim to have detailed information about the Internet architecture in order to know which routers to activate throttles on. The number of upstream routers that must participate in the flow control can become very large, and each must incur the overhead of counting how many packets they allow through to the receiver. It also requires an authentication mechanism for the flow control messages. Otherwise, an attacker could use the flow control messages to create a denial of service by setting up unauthorized throttles.

4. Distributed Filtering

Distributed filtering is a means of filtering packets similar to ingress filtering. A router agent that performs distributed filtering checks each packet to determine if its source address is valid based on the agent's knowledge of the Internet topology. For example, if a router receives a packet from host A on a link other than the one that traffic from A should arrive on, the packet will be dropped. It has been shown that a number of these agents, installed on only 20% of the autonomous systems in the Internet, would be sufficient to prevent up to %96 of all DoS attacks, and allow the location of a source of a successful attack to be narrowed to a set of five domains [PARK01].

The problem with this type of defense is that it relies on detailed knowledge of the Internet topology. Exchange of source routing information between routers imposes additional overhead and would require some type of authentication. The required cooperation of 20% of the autonomous systems on the Internet is also negative. Additionally, this method does not filter traffic with spoofed IP addresses that are still within a valid range for a given domain. If hosts A and B are both in the same domain, the countermeasure would be ineffective against packets sent from B but marked with A's address. Some type of ingress filtering would be required for attacks of this type.

5. Drawbacks

The main drawback shared by the above countermeasures is the need for cooperation from third-party routers or service providers. Obtaining this cooperation may be difficult or impossible based on the provider's unwillingness or inability to act, and linguistic and geographic barriers. The inability to stop attacks in a timely manner, requirements for changes to network devices and protocols, processing overhead, and requirements for detailed knowledge of changing Internet topology make it unlikely that these solutions will be implemented in a widespread and effective manner.

G. DOS ATTACKS IN A DIFFSERV ENVIRONMENT

New types of DoS attacks will accompany implementation of the DiffServ model. The separation of traffic into distinct classes and policing traffic on a per client basis will make it easier for an attacker to target a specific subset of the traffic flowing between nodes. Since resources for individual traffic classes will be limited, it may be easier to exhaust resources available to those classes. Furthermore, bandwidth limits imposed on sources in order to maintain QoS guarantees will impose an artificial bottleneck that attackers can exploit. If the DiffServ network reduces the bandwidth available to best effort traffic in order to maintain service guarantees to other traffic, it may inadvertently facilitate a DoS attack against best effort traffic. Service theft, the unauthorized use of guaranteed services, may also result in a denial of service to legitimate users of those services. This section describes the effects that DiffServ implementation will have on the conduct of DoS attacks.

If an attacker can compromise hosts or routers within the DiffServ domain, creation of a DoS network or direct interference for traffic flowing in the domain is trivial. Similarly, an attacker could deny service to a client by compromising client systems. Solutions to these types of attack are beyond the scope of this thesis. This research focuses on a method of attack similar to the most common types of attacks employed in the current Internet.

1. Attack Constraints

Indirect interference through bandwidth consumption will be harder to accomplish if flood traffic must traverse a DiffServ domain, since packets from paying clients will receive preferential treatment. Packets from unrecognized sources will be assigned a default code-point, and will not receive special treatment. Valid packets will be less likely to be dropped and more likely to reach their destination in a timely manner because of the priority they receive over best-effort traffic. Thus, implementation of DiffServ will inherently provide more DoS protection for traffic aggregation points. With servers less vulnerable, the overall effect will be a shift in the focus of DoS attacks away from traffic termination points and towards other points along the path.

The point at which traffic enters a DoS domain is the next logical point of attack. If a DiffServ provider is to use finite resources to provide QoS guarantees to all clients, it must limit the amount of traffic individual clients may send. The amount of traffic a client may receive premium service for is defined in the SLA with the DiffServ provider. Bandwidth limitations imposed by SLA enforcement will make the entry point of the DiffServ network the narrowest section of the path.

DiffServ domains only provide preferred service to recognized clients. At the network layer, incoming packets are classified to receive preferential treatment if their source address matches the address associated with an existing SLA. Consequently, DoS attacks will rely on spoofing the address of a valid DiffServ client to attack that client or the class of traffic associated with that client's SLA. This in turn requires that the attacker have some means of identifying the clients of a DiffServ provider. Typically, this will be accomplished through installation of some type of traffic monitor or "sniffer" along the path from the client to the DiffServ domain.

The DiffServ domain must be able to meter each client's traffic in order to ensure client adherence to SLA's with regard to usage amounts. It is possible to distribute this metering across all ingress routers, or to process and store metering information in a central database. However, it is assumed that this metering will not be distributed or managed in a centralized manner. Instead, the DiffServ domain assigns a specific ingress router as the designated entry point for traffic from a given client. This eliminates overhead associated with intra-domain metering communications. It also allows the DiffServ domain to filter incoming traffic based on the router it arrives at. Therefore, to conduct a successful attack, the attacker must not only spoof the address of a valid client, but it must ensure that flooding traffic arrives at the ingress router assigned to that client.

In a wired network, if a client is only one hop away from the ingress router, the DiffServ domain will be able to filter traffic based not only on the incoming router, but also on the incoming link. In this case, it will be impossible for an attacker to flood spoofed traffic using this client's source address, since we have already stated that the client itself is not compromised. It follows that no attack is possible unless the client is more than one hop away from its assigned ingress router.

The above reasoning does not hold if the client's connection to the ingress router is a wireless link. In a wireless connection, the transmission medium itself is not secure. Even if a client is only one hop away from the ingress router, an attacker could inject malicious traffic with that client's address, thus circumventing the interface filtering at the ingress router. Use of wireless networks also simplifies covert collection of information about clients, since packets are broadcast into an unprotected medium.

2. Countermeasures

DiffServ implementation will enable new methods for prevention of attacks without interfering with the effectiveness of existing countermeasures. Current attacks rely on the inability of the target to determine the source of flooding, thereby preventing the flood from being stopped or effectively filtered upstream of the aggregation point. However, since QoS guarantees are only provided to paying clients, the DiffServ provider must maintain a database of clients in order to properly meter traffic and provide appropriate QoS. The provider can use this data at ingress routers to quickly downgrade or drop packets marked with non-client source addresses. Of course, an attacker could simply forge the source addresses of actual clients, so the router must have another means of filtering malicious traffic. However, this requires the attacker to use addresses of hosts that the DiffServ provider knows are valid. This is a key benefit, since having a valid source address to contact allows verification of the authenticity of the traffic being received.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DENIAL OF SERVICE COUNTERMEASURE

A. ASSUMPTIONS

The majority of the assumptions made when developing this countermeasure were based upon the discussion in Chapter III, Section D. To summarize, client traffic must follow a partial best effort path. The attacker can observe the traffic, and direct some number of agents installed on compromised hosts to mimic the signature of the client's packets. These hosts then send a flood of traffic to the ingress router assigned to that client.

One other assumption was necessary to define the environment in which the countermeasure would operate. The bandwidth of the connection between the DiffServ client and provider is assumed to be greater than the combined bandwidth available to the attacker for flooding. If this were not the case, the attacker would not have to use a new type of attack, but could simply conduct a traditional indirect DoS against either the provider or the client.

B. IP AUTHENTICATION HEADER (AH)

The Authentication Header extension to the IP protocol is used to provide connectionless integrity and data origin authentication for IP datagrams [RFC2402]. The sender "signs" individual packets using a shared secret key, a hashing algorithm, and portions of the data contained in the packet. The receiver uses the same key and algorithm to verify the signature of the received packet. If the IP header is included when calculating the signature, the receiver can authenticate the source address field of the message.

It is possible for a receiver to determine the validity of a packet's source address for 100% of IP AH packets received. However, the per-packet cryptographic processing required for IP AH does not scale well, and may be too computationally intensive to implement while maintaining QoS guarantees. Ingress routers would be required to verify the header of every packet received before they could decide whether or not to discard it. Studies have shown that the maximum data rates of several widely used secure hashing algorithms are not sufficient for use in high-speed networks [RFC1810, RIPEMD96].

Another drawback of IP AH is that it requires the insertion of an additional header section into each packet. For small packets, such as those used for VoIP traffic, this additional header information represents a significant percentage of the overall size of the data packets.[1] The increased size would result in greater transmission delays, and could adversely affect the QoS a DiffServ provider could guarantee. For applications that use larger packets, such as UDP-based streaming video or audio, increasing the size of IP packets can result in fragmentation, which also negatively affects QoS.

C. GOALS

The goal of this research was to develop a DoS countermeasure that would be effective under the assumptions above. To be acceptable, the countermeasure

- must not require per-packet cryptographic processing,
- must not rely on cooperation from third-party hosts or routers,
- must not increase the likelihood of packet fragmentation, and
- must have negligible or no effect on the provider's ability to guarantee QoS.

The goal was not to devise a solution that would guarantee authentication of 100% of incoming packets or a zero loss rate for valid, in-profile traffic. Some degree of loss was deemed acceptable in exchange for reduced processing overhead.

D. CONCEPT

The DoS countermeasure that was developed is a marking method that a client can use to make its packets readily distinguishable from traffic with forged headers in certain cases. Supporting this design is a feedback mechanism through which a provider can notify a client when its traffic does not conform to the profile specified in its SLA.

This thesis proposes a technique that will allow the ingress router of the DiffServ domain to distinguish valid packets from malicious ones based on signature. A packet's signature is defined as a combination of the source address field and one or more other

[1] The minimum size of the AH header is 16 bytes; maximum size depends on the size of the encrypted packet and the AH options selected. The size of a common VoIP packet (without AH) is 64 bytes.

fields in the IP header. This method relies on the ability of the client to alter this signature. Since it has been assumed that an attacker will be able to observe traffic flowing between the client and the DiffServ domain, and will be able to instruct the flood sources to mimic any changes to packet signatures that it observes, changing the headers once is insufficient. Therefore, changes will be made on a periodic basis, and must be made faster than the attacker can duplicate them.

Figure 3.1 is a time diagram showing the sequence of actions involved in the proposed countermeasure. When the ingress router marks a packet that appears to originate from a DiffServ client as out-of profile, it will log the source and time (t_1) of the drop. When the rate of out-of-profile marking exceeds a pre-set threshold (t_2), the router will send a feedback message (A) to the client. Upon receipt (t_3), the client will begin altering the signature of its packets. The ingress router will use these alterations to identify valid packets. It will drop all packets with an invalid signature. Details of the individual actions taken by the client and DiffServ router are given below

The router feedback to the client will consist of a router-generated seed key for an algorithm that generates a sequence of signatures. The client and router will be able to independently calculate what the correct signature should be using this algorithm and the seed-key. The algorithm can be well known as long as the seed key being used remains secret. The seed key will be encrypted using a shared secret key and digitally signed. A seed key is used to generate new signatures instead of the shared secret key to avoid compromising the secret key through overuse. The digital signature provides authentication for the feedback message, so attackers will be unable to create a DoS by forging these messages. Payload encryption is required since the attacker can monitor traffic flowing between the client and the DiffServ domain. Since the algorithm for generating signature values is not secret, access to unencrypted seed keys would allow the attacker to change the signature of the attack packets as rapidly as the sender could, thus circumventing the countermeasure.

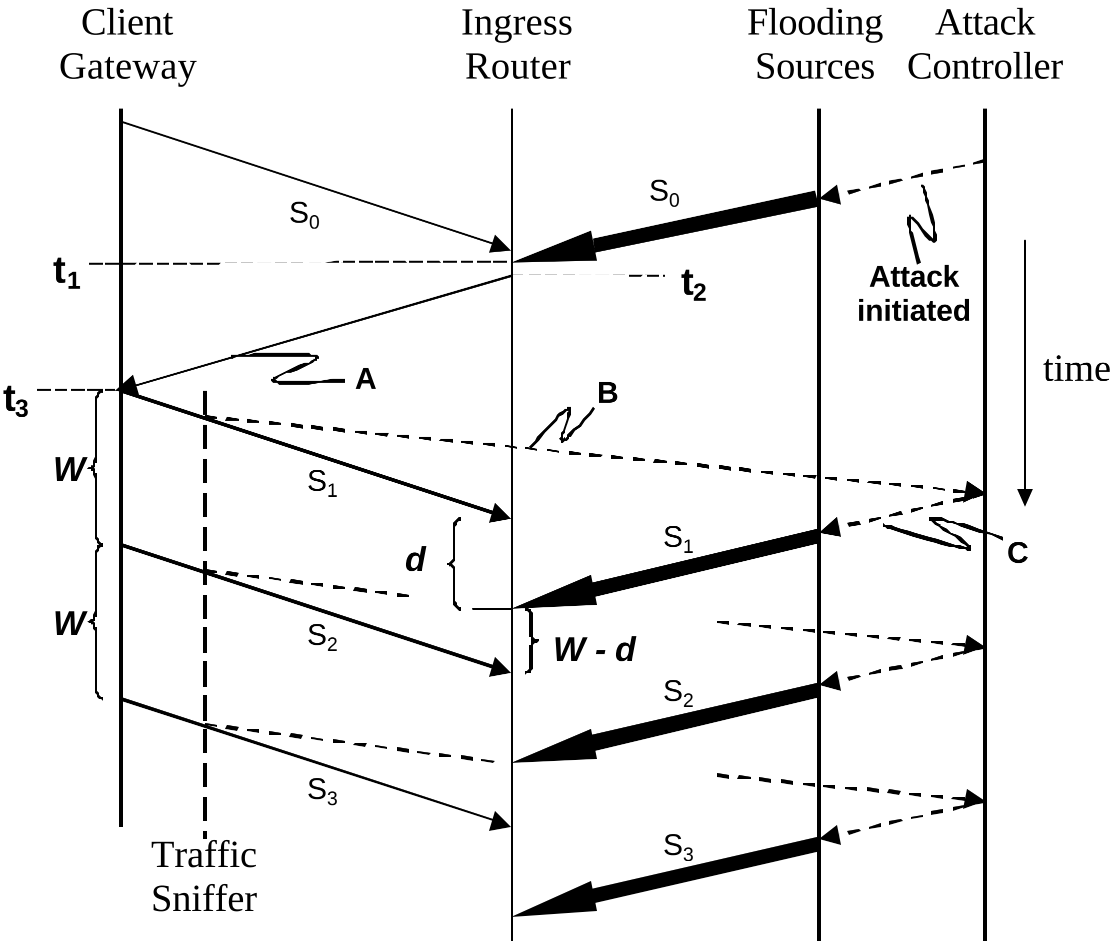

Figure 3.1. Time sequence Diagram for DiffServ Countermeasure

Upon receipt of a feedback message, the client will authenticate it, decrypt the payload, and use the seed key to calculate the sequence of signature values that it will use. The client will immediately begin using the values in the designated fields of the IP headers of its packets. It will switch to the next signature (S_i) in the sequence at regular intervals denoted by *W*. The attacker will not know what each new signature is until it receives the information from the monitor installed along the path of the client traffic (B). When it knows the new signature, it can direct the flood sources to change the signatures they are using (C). The time between when the ingress router receives the first valid packet with a new signature and when it receives the first attack packet with the same signature, denoted by *d*, is the window in which 100% authentication is possible. The importance of the relative values of *d* and *W* are discussed in Section D.

The seed key is also used to create the same sequence of signature values at the router. After sending a feedback message, the router will treat all packets as valid until it receives the first packet with the first altered signature. All successive packets with an incorrect signature are dropped, except the first packet received with the second signature. When the first packet with the second signature is received, the router will drop all successive packets that do not match the second signature, including those marked with the first signature**.** This prevents an attacker from using old signatures to circumvent the DoS countermeasures.

E. THEORETICAL PERFORMANCE

The main performance metric of interest is the client's *packet out-of-profile rate*, i.e., the percentage of the DiffServ client's packets being marked out of profile at the ingress router. A DoS countermeasure is considered more effective than another is if it achieves a smaller packet out-of-profile rate for the client given the same network setup and attack scenario. An analytical model of the feedback mechanism was created using a set of simplifying assumptions. From this model a closed form solution for the DiffServ client's packet out-of-profile rate was derived.

Denote the percentage of the client's packets being marked out of profile at the ingress router by *p*. In order to derive *p*, the following additional assumptions were made:

1) All client and attack packets are the same size.

2) The client's traffic arrives at the ingress router at a constant rate of *r* packets per second, which is less than or equal to *CIR*, the client's committed information rate in packets per second.

3) The attack traffic arrives at the ingress router at a constant rate of *A* packets per second such that

$$r + A \geq CIR \quad . \tag{1}$$

This means the percentage of valid traffic received by the ingress router is equal to

$$\frac{r}{r+A}. \tag{2}$$

4) The client traffic switches to a new signature every W seconds. The attack traffic tries to make the same signature change, but the change always happens *d* seconds later from the ingress router's perspective. In other words, there is a fixed lag of *d* between the arrival time at the ingress of the first valid packet with a new signature and the arrival time of the first attack packet with the same signature.

5) The ingress router's traffic metering process for the client is fair so that if the traffic being metered is made of several flows, each flow will be ensured of a share of in-profile packets that is proportion to the flow's packet arrival rate.

Consider the time window for an arbitrary signature used by the client's traffic. There are two cases:

Case 1: $W \le d$. From assumption 4, during the entire time period, every attack packet carries an expired signature when inspected by the ingress router. Such packets will be dropped before being counted against the client's committed rate in the metering process. From assumption 2, the rate of the valid traffic alone does not exceed the committed rate. Thus, we have $p = 0$.

Case 2: $W > d$. From assumption 4, during an initial time period equal to *d*, the ingress router will be able to drop all attack packets. However, for the remaining time $W - d$, the ingress router will not be able to distinguish valid traffic from attack traffic because they have the same signature. In that case, some of the client's packets will be marked out-of-profile. From assumption 5 and equation (2), the percentage of client packets marked as in-profile during this period is

$$[(W - d) \cdot CIR]\frac{r}{r + A}, \tag{3}$$

so the number of client's packets marked out of profile during this period is:

$$r \cdot (W - d) - [(W - d) \cdot CIR] \frac{r}{r + A} \tag{4}$$

$$= (W - d) \cdot r \times (1 - \frac{CIR}{r + A}) . \tag{5}$$

Dividing equation (5) by the total number of packets sent by the client during the entire time window, $r \cdot W$, yields

$$p = (\frac{W - d}{W}) \times (1 - \frac{CIR}{r + A}) . \tag{6}$$

It can be shown via a similar derivation that p_0 , the packet out-of-profile rate for a client that does not use any DoS countermeasure, is equal to

$$p_0 = (1 - \frac{CIR}{r + A}) . \tag{7}$$

Using equation (7), we rewrite equation (5) as:

$$p = (1 - \frac{d}{W}) \times p_0 . \tag{8}$$

Equation (8) clearly indicates that the reduction in the packet out-of-profile rate due to the feedback mechanism is inversely proportional to *W*, the period between signature changes by the client.

Combining both cases results in the following theorem:

Theorem 1. After a client initiates the DoS, the client's packet out-of-profile rate becomes

$$p = \max\{0, (1 - \frac{d}{W}) \times (1 - \frac{CIR}{r + A})\} . \tag{9}$$

F. WEAKNESSES

A rudimentary implementation of this countermeasure would nave several weak points. This section discusses several known weaknesses of the DoS countermeasure and possible solutions.

1. Size of Signature Space

One weakness is based on the size of the signature space, which is defined as the combined total number of bits in the IP header fields used as the signature. If the signature space is small, an attacker can launch an attack such that packets containing every possible signature are sent to the DiffServ ingress router during any given signature window *W*. As a result, the ingress router will receive attack packets with signatures matching the next expected signature before the client has switched to that signature. Processing these packets will trigger a premature signature change. The resulting signature asynchrony between the client traffic and the ingress router will cause all future client packets to be dropped by the DoS countermeasure. This is a more effective DoS than if the countermeasure was not in use.

Even if the signature space is sufficiently large, an attacker may be able to send enough packets with different signatures to increase the statistical likelihood that one of them will have the correct next signature. To counter this, the client can also use the feedback mechanism

2. Path Variations

The client's ability to change its signature faster than an attacker can duplicate the changes is not guaranteed. It may be possible for attack traffic with the next signature to arrive at the ingress router faster than valid traffic with the same signature. As stated above, *d* depends on the difference in the ingress router arrival times of valid and invalid packets with the same signature. If the difference is equal to zero, then from the ingress router's point of view, the attacker can match the signature changes as fast as or faster than the client can make them and the countermeasure is useless. If the difference is negative (i.e. attack packets arrive faster than valid packets), the countermeasure actually aids the attacker, since valid packets with the previous signature will be dropped after receipt of the first attack packet with the new signature.

3. Traffic Re-ordering

If valid traffic is re-ordered at some point before reaching the ingress router, the countermeasure may have an adverse effect on valid traffic. A packet containing a new signature that is received out of order, i.e. ahead of packets with the old signature, will cause in-transit packets with the old signature to be dropped. This can be remedied by delaying the expiration of old code-points. The length of the delay can be based on the mean propagation delay incurred by client traffic as observed by the ingress router. However, this remedy reduces the effective period of the countermeasure, since flooding traffic with the old signature will also be accepted during the overlap period.

4. Fairness

The countermeasure may disproportionately drop packets from certain sub-flows within the clients traffic. Recall the steps for deriving the client's out-of-profile rate. If *W* is a constant and if *W* is larger than *d*, the DoS countermeasure creates periodic DoS effective time intervals as illustrated in Figure 4.1. During these time intervals, the countermeasure is ineffective and the DoS attack causes high packet out-of-profile rates for the client. If one of the client's sub-flows generates packets periodically, it is possible that the flow started during a DoS effective interval and its packet generation period is similar to *W*. In such a case, that sub-flow's packets always arrive at the ingress during DoS effective periods, resulting a disproportionately high out-of-profile rate for the sub-flow.

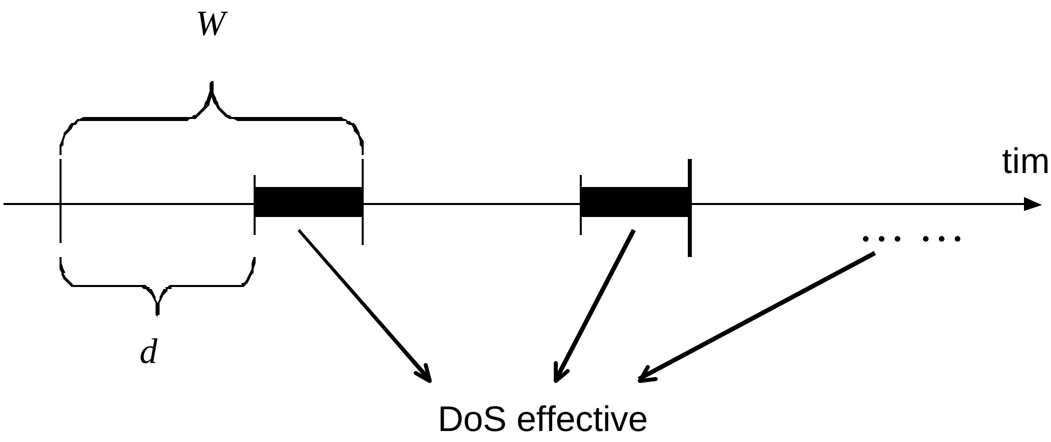

Figure 4.1. Periodic DoS Effectiveness

Therefore, the DoS countermeasure may not all sub-flows equally or fairly. One fix is to have the client randomize the value of *W*. We intend to evaluate the performance of this fix and other solutions to enhance the fairness of the countermeasure in our follow-on work

G. ADDITIONAL BENEFITS: DETECTION OF SERVICE THEFT

A service theft can be defined as a course of actions taken by a perpetrator to use a portion of a valid client's allocated bandwidth and obtain a premium service without pay. An intruder may attempt to blend its unauthorized traffic in with valid traffic by studying the client's traffic pattern and adjusting his own traffic volume over time so that the ingress router would never mark an excessive number of packets out of profile. Unlike a DoS attack, the perpetrator is typically much more restrained to avoid drawing attention to himself. Therefore, a service theft usually does not cause as much direct harm to the client as a DoS attack. However, service theft results in lost revenue and network availability for the DiffServ provider.

The proposed feedback mechanism and the resulting cooperation between the client and the ingress router may help the service provider detect service theft. For example, the ingress router may activate the feedback mechanism randomly, regardless of whether or not it has just marked a large number of packets out of profile for the client. If a service theft is under way, the ingress router should notice two or more signatures being frequently used at the same time. When this occurs, the ingress router may log the event as a possible occurrence of service theft or immediately alert the network operator to perform further investigation. Use of a random signature time window (*W*) would prevent the perpetrator from predicting how much longer a spoofed signature would be valid and adjusting its traffic pattern accordingly.

If the intruder can monitor feedback messages from the ingress router, he can evade the ingress router's service theft detection by suspending his flooding traffic whenever a feedback message is observed. In this case, the proposed theft detection mechanism may not be able to catch a more resourceful intruder. However, it will be sufficient to stop the service theft from continuing, and more importantly, it will deter future service theft by forcing the intruder to expend more resources and effort to avoid detection.

The general conclusion that can be drawn from the above discussion is that the proposed DoS countermeasure may be extended into an auditing function that is orthogonal to access control. For example, it may be used to supplement some other DoS countermeasure by detecting service theft or cases where the authentication process of the other countermeasure has been compromised.

THIS PAGE INTENTIONALLY LEFT BLANK

V. SIMULATION

A. NS2 NETWORK SIMULATOR

The experiments were conducted using version 2.1b8a of ns2, a discrete event simulator targeted at networking research [NS02]. The following description of the simulator is given in *The ns Manual*:

> *ns* is an object-oriented simulator, written in C++, with an OTcl interpreter as a front end. The simulator supports a class hierarchy (also called the compiled hierarchy …), and a similar class hierarchy within the OTcl interpreter (also called the interpreted hierarchy …). The two hierarchies are closely related to each other; from the user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy. … Users created new simulator objects through the interpreter; these objects are instantiated within the interpreter, and are closely mirrored by a corresponding object in the compiled hierarchy. [FALL02]

The simulator includes a DiffServ module with implementations of distinct core and edge routers, several marking policies and queuing disciplines, and built-in tracing for DiffServ queues.

The remaining material covered in this section is drawn from the detailed description of ns2 in [FALL02]. In the remainder of this thesis, names of objects and functions in the compiled (C++) hierarchy are in **`Courier bold`**. Names of objects and functions in the interpreted (OTcl) hierarchy are in *italics*. By convention in ns, subclasses are denoted by specifying the name of the parent class followed by a forward slash and the name of the subclass. For example, *Agent/UDP* is a subclass of *Agent*, and *Agent/UDP/Sniffer* is a subclass of *Agent/UDP*.

1. Interpreted Objects

The simulator is configured using OTcl scripts to define the parameters of network devices and interrelationships between them. Objects in the script are instantiated in the compiled and interpreted hierarchies when the simulator is initialized at run time. This subsection describes the OTcl objects necessary to construct and run an ns2 simulation.

Most scripts begin by instantiating a *Simulator* object, which represents an instance of the ns2 simulator. This object provides interfaces for simulator configuration and for setting up the type of event scheduler to be used. To instantiate simulator objects and configure other aspects of the simulation, various methods of the class *Simulator* are called.

Node objects represent network hosts and routers. A *Node's* address may be assigned automatically or manually. Unicast nodes contain two *Classifier* objects, which determine how packets are forwarded. The address classifier determines whether a packet is addressed to the node. If it is not, the packet is passed to the appropriate *Link* object, as determined by the routing algorithm, for forwarding. If the packet is addressed to the node, it is passed to the node's port classifier. The port classifier passes the packet to the *Agent* object attached to the port specified in the packet's header.

A *Link* object connects two nodes and contains the mechanisms for simulating packet queues and propagation delays. Links may be unidirectional (*simplex-link*) or bi-directional (*duplex-link*). Every link has a *Queue* object associated with each traffic flow direction. A queue is the location where packets are stored until they can be forwarded to the link's destination node. The simulator provides a number of disciplines for queue management and scheduling.

Agent objects are used to implement network protocols or services. Agents are attached to nodes on a specific port, and they receive all traffic destined for that port. Similarly, they are the source of packets originating from that port. Agents may manage packets based on native code, or they may have an attached *Application* object that determines how outgoing packets are generated and incoming packets are processed.

Two agents can be linked using the *connect* command. This designates each agent as the destination for traffic from the other agent. When one of the linked agents creates a packet, the destination address is automatically set to the address of the other agent. The *Agent* subclass *Null* implements a sink for traffic that requires no processing at the destination.

The *Simulator*, *Node*, *Queue*, *Agent,* and *Application* classes are mirrored in the C++ hierarchy by classes of the same name. The *Link* class exists only in the OTcl hierarchy.

2. Compiled Objects

The exchange of references to Packet objects between simulator objects is used to simulate packet flow in the simulator. The user can specifically choose which protocol-specific headers are included in simulation packets, e.g. TCP, UDP, TELNET, and ARP. By default, every Packet object includes the structures for every protocol-specific header defined in ns2. New header structures can be added by defining a C++ structure with the required fields, creating an OTcl linkage for the structure, and modifying the simulator initialization code to include the new header. Class **`Packet`** is not mirrored in the OTcl hierarchy, since individual packets are not accessed in the interpreter.

The struct **`ns_addr_t`** is defined in the file config.h, which resides in the ns-2.1b8a subdirectory of the main ns2 directory. This struct is a container for an address and port number pair. It is used to correctly route traffic to or identify traffic from the Agent residing at the given port on the *Node* with the given address.

The abstract base class **`TimerHandler`** provides the C++ means of scheduling future events. This class is also not mirrored in the OTcl hierarchy. Classes derived from **`TimerHandler`** must define the pure virtual function **`expire()`**. This function definition is the action that will be taken when the event is executed. Based on the input parameter they receive, the **`sched()`** and **`resched()`** functions will schedule an event a certain number of seconds into the future. At the scheduled time for the event, the simulation calls the **`expire()`** function, which executes the event. The user can schedule a recurring event by calling the **`resched()`** function inside the **`expire()`** function.

3. DiffServ Implementation

The DiffServ module in ns2 allows the user to define a number of different behavior aggregates and their associated PHBs. Packets arriving at an ingress point are marked with the code-point associated with their behavior aggregate. DiffServ routers provide varying QoS by applying different forwarding treatment to each aggregate. DiffServ objects also automatically maintain records of dropped packets.

DiffServ routers in ns2 use Multiple Random Early Detection (MRED) for queuing management. RED is designed to prevent overflow by randomly dropping packets when a queue's usage exceeds a certain percentage of its maximum capacity. MRED applies a RED algorithm to a group of queues, but the drop probability for traffic assigned to an individual queue may be affected by the average length of all queues in the group.

The class **`dsREDQueue`** is derived from class **`Queue`**. It provides a single DiffServ router the functionality to handle multiple RED queues on a single link. Two classes representing DiffServ edge (ingress) and core routers, **`edgeQueue`** and **`coreQueue`**, are derived from **`dsREDQueue`**. They are mirrored in the interpreted hierarchy as *Queue/dsRED/edge* and *Queue/dsRED/core*, respectively.

The class **`Policy`** provides the functionality to police traffic into different aggregates. It also maintains a policy table with the parameters for each source and destination pair. The **`mark()`** function assigns an initial code-point to the arriving packets of a flow based on their source and destination addresses, and then calls the policer via the **`applyPolicy()`** function. The policy modules currently implemented in the DiffServ module are Time Sliding Window with Two or Three Color Marking, Single or Two Rate Three Color Marking, and Token Bucket. The experiments conducted for this research used the Token Bucket policer, which is covered in detail below. The policer may downgrade the initial code-point to a secondary or tertiary code-point based on the state information maintained in the policy table for the flow.

The Token Bucket policer increments the policy's **`cBucket`** state variable according to the elapsed time since the arrival of the last packet. The variable **`cBucket`** is incremented at a rate equal to the policy's CIR, but is capped at an upper bound equal to the policy's committed burst size (CBS). If the arriving packet's size is less than or equal to **`cBucket`**, **`cBucket`** is decremented by the packet size and the packet retains its initial code-point. Otherwise, the packet's code-point is downgraded to the secondary code-point.

B. DESIGN CONSIDERATIONS

The DoS countermeasure was designed to require a minimum number of changes to existing ns2 files. When possible, existing C++ classes were extended to add new functionality. In all cases where this was not feasible, backwards compatibility was maintained with the version of ns2 being modified. This was accomplished by adding new functions that are called only by code associated with the countermeasure, or by use of conditional statements that are executed only if the DiffServ module is installed. Existing lines of code were not modified.

Since OTcl procedures do not need to be defined in the same file as their parent classes, new OTcl procedures were added in a separate file, and the compiler was reconfigured to include this file during building of the ns2 executable.

C. IMPLEMENTATION

Simulation of the DoS attack and countermeasure was implemented through modification of existing ns2 classes or by creation of new classes. This section describes the changes and additions to the simulator. The code for these changes can be found in Appendices A and B.

1. Simulator Extensions

The following classes were added to the ns2 Simulator. They provide the methods and objects necessary to store information about the feedback process, to communicate between the DiffServ client and provider, and to create a simulated network topology based on the constraints described in Chapters III and IV.

a. Class `dsFeedback`

The DoS countermeasure is primarily implemented in class **`dsFeedback`**. All members of this class are static, which allows other classes to access them without instantiating the parent class, and insures that only one state information database exists. Member functions of this class enable database access and modification; generate code-points used by the countermeasure; and initiate of feedback messages.

Class **`dsFeedback`** maintains a state information database for each source and destination pair seen by the ingress router. The upper level of the database is a C++ Standard Template Library (STL) **map**. This **map** pairs each source address with a second **map** object. The second **map** pairs destination addresses with objects of class **FeedbackInfo**. Each **FeedbackInfo** object contains the state information for the specified source and destination pair. The state information includes

- an STL **deque** of packet drop times,
- a flag indicating whether the countermeasure is running or not,
- the current and next code-points to be used by the source,
- the length of the key window, and
- the parameters which determine the drop rate which, if exceeded, will result in a feedback message

Class **IcmpAgent** is declared as a friend of class **dsFeedback**. This allows it to access the private member variable **secretKey.** This variable represents the shared secret required for authentication and decryption of feedback messages, and code-point sequence generation.

The source code for classes **dsFeeback** and **FeedbackInfo** is in the files dsFeedback.{h,cc}, which are in the /ns-2.1b8a/diffserv directory. These classes are not mirrored in the OTcl hierarchy.

*b. Class **IcmpAgent***

The **IcmpAgent** class is derived from class **Agent**. It provides the mechanisms for sending feedback messages and scheduling client code-point changes. The struct **hdr_icmp** defines the ns2 header associated with ICMP agents. It contains information about the source of a dropped packet, the seed key to be used to generate the code-point sequence, and the code-point window to be used by the client.

The member function **send()** overrides the version in the parent class. The ingress router calls this function when the drop rate exceeds the threshold. The function creates a new packet, assigns the appropriate value to the fields in the ICMP header, and sends this feedback packet to an ICMP agent attached to the source of the dropped packets.

The member function **recv()** also overrides the implementation in the parent class. It is called when the parent node of the ICMP agent receives a feedback packet. The function uses information in the packet's header to configure and start the DoS countermeasure.

Each ICMP agent contains an instance of class **CodePtTimer**, a timer derived from **TimerHandler**. The **expire()** function of **CodePtTimer** calls the static **generateNextCodePt()** function of class **dsFeeback** and then reschedules itself based on the code-point window specified in the feedback packet from the ingress router.

The source code for classes **IcmpAgent** and **CodePtTimer,** and the **hdr_icmp** struct may be found in the /ns-2.1b8a/diffserv directory in files icmp.{h,cc}. **IcmpAgent** is mirrored in the OTcl hierarchy as class *Agent/Icmp*.

*c. Class **SnifferAgent***

The class **SnifferAgent** is used to simulate a monitor installed along the path of traffic coming from the client gateway. This class is derived from the existing **UDPAgent** class, and is mirrored in the OTcl hierarchy as class *Agent/UDP/Sniffer*. The member variable **controller** is a reference to a **ControlAgent** object attached to a separate node in the network. This other node is the location of the attack controller. Whenever the source being sniffed sends a packet with a changed code-point, an additional packet is sent to the agent that **controller** refers to. This simulates the ability of the attack controller to monitor traffic from the source and 'see' code-point changes. The source code for **SnifferAgent** is contained in /ns-2.1b8a/diffserv/sniffer.{h,cc}.

*d. Class **ControlAgent***

This class is also derived from **UDPAgent**, and is mirrored as *Agent/UDP/Controller*. It simulates the DoS attack controller. The member variable **floodList** is a reference to a list of flooding sources. The overridden function **recv()** examines the code-point in packets sent from a **SnifferAgent** and changes the code-point used by attack traffic to match it.

Due to problems encountered with spoofing addresses in the simulator, changes to the code-point used by the compromised hosts could not be made by sending a packet containing the new code-point from the controller to the flood source. Instead, the code-point used by the flood sources was directly manipulated by the attack controller. This introduced an artificial acceleration of the attackers ability to mimic the clients code-point changes, since queuing, transmission, and propagation delays between the controller and flooder were eliminated. However, the validity of the simulation was maintained by increasing these delays between the flooding sources and the DiffServ ingress router in the simulation script. The source code for **`ControlAgent`** is contained in /ns-2.1b8a/diffserv/attack-controller.{h,cc}.

e. Class `FloodAgent`

This class simulates the flooding sources controlled by an attacker. It is derived from **`UDPAgent`**. The ability to alter the code-point in use upon receipt of a packet from the attack controller was implemented, but was not used for reasons discussed in the section on **`ControlAgent`**. The source code for **`FloodAgent`** is contained in /ns-2.1b8a/diffserv/zombie.{h,cc}.[2]

f. OTcl Procedures

Addition of three OTcl procedures was necessary to allow the exchange of certain data between the compiled and interpreted hierarchies. These procedures are located in the file /ns-2.1b8a/tcl/lib/ns-diffserv.tcl. The procedures *prio* and *set_priority* were added to class *Agent* to allow the user to view and modify an agent's code-point for debugging purposes. The procedure *get-agent-handle* was added to the ICMP agent class (*Agent/ICMP*). It takes a port number as an argument and returns a reference to the agent attached to that port. The procedure allows an ICMP agent in the C++ hierarchy to obtain a reference to another agent attached to the same node.

2. Simulator Modifications

The following existing ns2 files were modified to implement functions necessary for either creating the attack scenario described in Chapter III or implementing the DoS

[2] For consistency with the thesis text, simulator files zombie.{h,cc} are renamed flood.{h,cc} in the appendices, and the word "zombie" has been globally replaced with the word "flood"

countermeasure described in Chapter IV. Modifications to existing files can be found quickly by searching for the flag "[FEEDBACK]".

a. Class `dsred`

The class **`dsred`** simulates queue objects in DiffServ enabled routers. This class is mirrored in the OTcl hierarchy as class *Queue/dsRED.* It contains a struct **`stats`**, which stores data about the number of packets dropped. The **`enque()`** function of this class takes a **`Packet`** object as an argument and queues or drops it based on the algorithm chosen by the user during configuration of the simulator.[3]

The member variable **`drops_FB`** was added to the **`stats`** struct to allow tracking of the number of packets dropped by the DoS countermeasure. The **`enque()`** function was modified in two ways. First, prior to performing any metering, incoming packets are checked to see if they are marked with an invalid code-point. If the code-point is invalid, **`drops_FB`** is incremented, and the packet is dropped without further processing. Otherwise, the packet is processed by the pre-existing code. The second change was inserted just prior to the points at which packets are dropped during normal processing. The simulator checks to see if the countermeasure is active at the source of a packet about to be dropped. This is done to ensure that no excess overhead is incurred by sending feedback messages to a source that is already running the countermeasure. If the countermeasure is not active, the record of drops for that packet's flow is updated in **`dsFeedback::feedbackTable`**.

b. Class `dsPolicy`

The class **`dsPolicy`** performs the metering functions of the DiffServ ingress routers. It is not mirrored in the OTcl hierarchy. The **`mark()`** member function of this class uses the meter and parameters specified by the user to set the code-points of packets arriving at the router. The **`mark()`** function was modified to call the **`dsFeedback::isCodePtValid()`** function prior to executing the existing code. Packets with invalid code-points are reassigned the code-point

[3] Choices for the queuing algorithm are drop tail, RIO C, RIO D, or WRED. A description of each can be found in the ns2 users manual [FALL02].

dsFeedback::INVALID_CP, and this code-point is also returned to the calling function. Packets with valid code-points are handled by the pre-existing code.

c. Class ***Agent***

The class **Agent** is the base class for all simulator agents. It is mirrored in the OTcl hierarchy as class *Agent*. The **Agent** protected member variable **prio_** represents the IP ToS field. The public member function **set_priority()** was added to allow modification of **prio_** outside of the **Agent** class.

d. Class ***Packet***

The class **Packet** stores a list of the headers included in a packet object in ns2. It is not mirrored in the OTcl hierarchy. The list was modified to include the header associated with the **IcmpAgent** class.

e. OTcl Packet Configuration

The file /ns-2.1b8a/tcl/lib/ns-packet.tcl contains the code necessary to enable and disable individual packet headers from within an OTcl script, as discussed in Chapter IV. This file was modified to include the header defined for use with the new ICMP protocol.

f. OTcl Default Parameters

The file /ns-2.1b8a/tcl/lib/ns-default.tcl stores default parameters that are assigned to OTcl objects when they are instantiated. The default value for ICMP packet size was added to this file.

g. Compilation Environment

The files Makefile.in and ns-lib.tcl are used when building the ns2 executable. Makefile.in describes the dependencies among the ns2 source files and contains the commands necessary to update the ns2 executable file. The location and names of all new C++ and OTcl files were added to this file. The names and locations of OTcl files that must be compiled into the executable are contained in ns-lib.tcl. This file was updated to include the file ns-diffserv.tcl.

VI. PERFORMANCE EVALUATION

A. EXPERIMENTAL DESIGN

Figure 6.1 shows the ns2 topology that was used to conduct experiments. The OTcl script that sets up this topology is contained in Appendix C. All hosts and intermediate routers are basic *Node* objects. All links external to the DiffServ domain consist of duplex *Link* objects with default *Queues*. Within the DiffServ domain, connections between ingress and core routers consist of two simplex *Link* objects. The links from ingress to core routers use *dsRED/edge* queue objects. The queue at the ingress router uses a Token Bucket policer to assign code points to incoming packets. The links from core to ingress routers use *dsRED/core* queue objects.

One *Sniffer* and one *Icmp* agent were attached to the client gateway node. The *Sniffer* agent was connected to a *Null* agent attached to the destination node. The *Icmp* agent was connected to a second *Icmp* agent attached to the ingress router node. An application that generated constant bit-rate (CBR) traffic (class *Application/Traffic/CBR*) was attached to the *Sniffer*

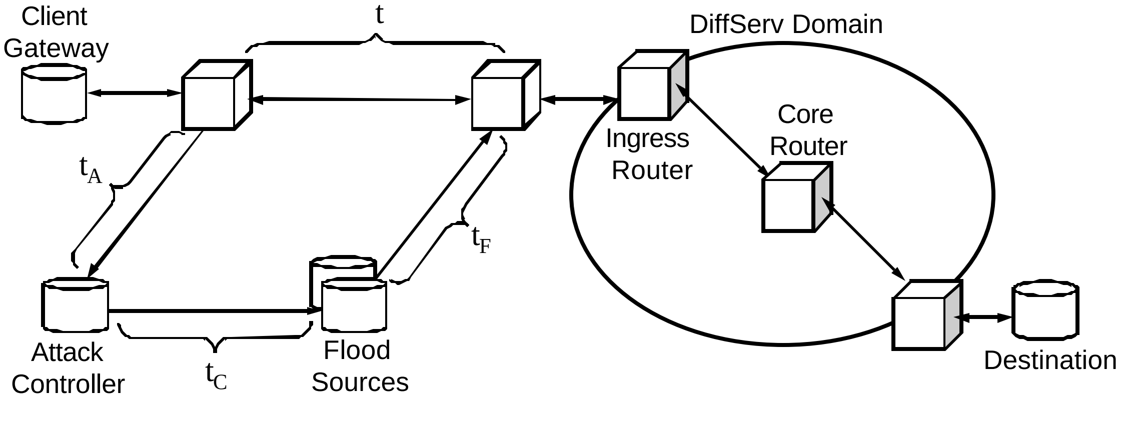

Figure 6.1. Experimental Network Topology

One control agent was attached to the attack controller node. A reference to this agent was stored in the sniffer agent on the gateway. One flood agent was attached to each flood source node, and each flooding agent was connected to the null agent installed on the destination node. References to the agents were stored in the list maintained by the control agent. The addresses of the flooding nodes were manually changed to match the address of the client gateway in order to simulate the spoofing of source addresses.

For all simulations, the client and flood sources were set to transmit fixed sized packets at a constant bit rate. A small degree of random variation in packet inter-departure times was introduced to eliminate synchronization of packet arrival at the ingress router. Link bandwidths were held constant, and were assigned values large enough to prevent queues from overflowing.

In Figure 6.1, the times t , t_A , t_C , and t_F represent the sum of all processing, queuing, transmission, and propagation delays incurred by a packet transiting the respective link. We observe that the difference in arrival times at the ingress router of the first valid and invalid packets with the same signature, which we have previously named *d*, can be written

$$d = \left(t_A + t_C + t_F\right) - t \quad . \tag{11}$$

As explained in Chapter V, Section B, the attack controller directly accesses the flood sources to update code points, thus eliminating the delays on the links between them. For this topology, this is equivalent to setting t_C equal to zero. However, this artificiality can be corrected by increasing the delay assigned to either t_A, t_F, or both, such that the sum of t_A , t_C , and t_F represents the total time required for the flow of attack packets with a new signature to merge with valid flows with the same signature.

After examining the IP header format, the Type of Service (ToS) field in the IP header was chosen as the mutable portion of the packet signature in our simulation. This field is already included in the simulation's IP header implementation, so no modifications to the header were required. The ToS field is unused in the non-DiffServ routers between the client and the DiffServ provider, so modifying it at the source will not affect packet routing outside of the DiffServ domain.[4]. DiffServ ingress routers change this field after receipt based on the client's SLA, so modifying it will not affect

[4] In practice, this field would not be used exclusively, since it may be used in transit by networks that implement IP Precedence [RFC791].

routing within the DS domain. In the remainder of this section, the term signature refers specifically to the combination of IP source address and IP ToS field. However, other fields such as ID or Options could be used in place of or in combination with the ToS field to determine packet signature.

B. EXPERIMENTAL RESULTS

In our first set of experiments, we compared the out-of-profile rate produced in the simulation to that calculated using Equation (9). Runs were conducted for several different values of p_0. The values of W, and p_0 were held constant during each run. The value of d was manipulated by varying t_A while holding t_C, t_F, and t constant. The results of these trials are shown in Figure 6.2.

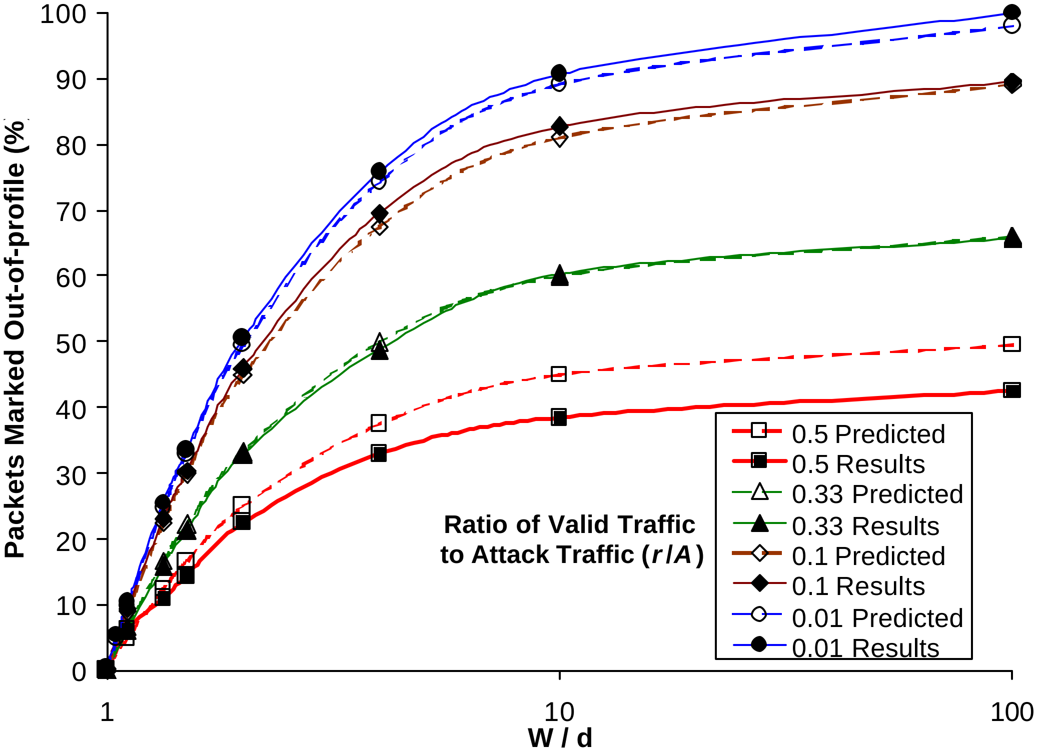

Figure 6.2. Predicted Out-of-Profile Rates vs. Measured Results

The second set of experiments was conducted to determine the effect on the DoS countermeasure of changing the bucket size for the Token Bucket policer. In each run, the values of W, d, and p_0 were held constant. The size of the token bucket was increased exponentially until it was large enough to prevent any packets from being dropped regardless of their true source. Runs were conducted for cases in which *W*>*d* and *W*<*d*. The results are plotted in Figure 6.3.

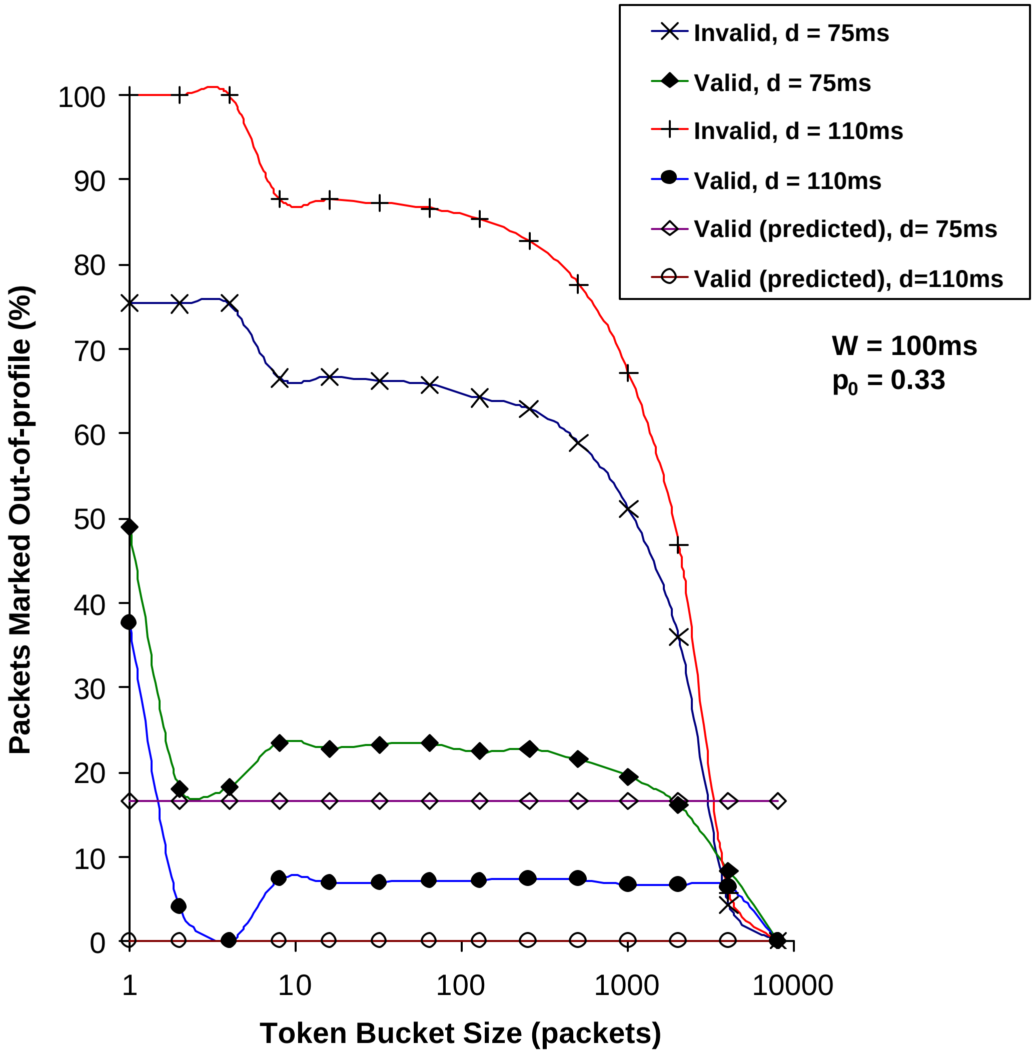

Figure 6.3. Effect of Token Bucket Size on Out-of Profile Rate

C. ANALYSIS

For the first experiment, the simulated results correlated well with the predicted results. For cases in which *W* was greater than *d* but the difference between them was small, the countermeasure was effective in limiting the out-of-profile rate for valid packets. When *W* >> *d*, the out-of-profile rate for valid packets approached p_0. In test cases where *W* < *d*, no valid packets were dropped as a result of the countermeasure.

Results of the second experiment indicate that for our countermeasure, a small token bucket size is required to minimize out-of-profile marking for valid traffic while maximizing it for invalid traffic. Bucket sizes that were two to four times the average packet size provided the best results. However, the result of failure to use a small bucket size was only slight degradation in the overall performance of the countermeasure.

The out-of-profile marking rate for valid traffic is higher than predicted if the token bucket size is not optimized. For extremely small token bucket sizes, this is expected. The token bucket allows the maximum arrival rate a router will tolerate for short periods to be larger than the average long-term rate. If the bucket size is small (less than twice the size of an average packet), the randomness introduced into inter-packet departure times can cause two successive packets to be received at a rate that exceeds the maximum arrival rate.

The observed effectiveness of the countermeasure was also worse than predicted for large bucket sizes. This can be attributed to the longer delay in starting the countermeasure that logically accompanies a larger token bucket. The ingress router will treat the initial flood of traffic as a burst. A larger bucket allows larger bursts, so under these conditions, it will take longer for packets to be marked out-of-profile once an attack commences.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSIONS

A. SUMMARY

DiffServ implementation will have a significant impact on the conduct and mitigation of DoS attacks. The explicit bond between a service provider and its clients will allow communication that is difficult to achieve in the current Internet. This research has demonstrated that it is possible to mitigate DoS attacks against DiffServ clients and detect service theft without per-packet cryptographic processing. The tradeoff for the reduction in overhead is the potential inability to guarantee a zero loss rate for valid traffic entering the DiffServ domain. The proposed countermeasure can be combined with other security protocols if both QoS and security are required. Its low cost in terms of processing requirements makes it an excellent choice as an independent monitor for possible breaches of the security protocols.

B. FUTURE WORK

The countermeasure and its implementation in ns2 are both in nascent stages of development. The following areas provide opportunities to extend and improve upon the work of this thesis.

1. Countermeasure Hardening

The proposed countermeasure has been exhaustively analyzed to ensure security. However, it has several known weaknesses (see Chapter IV) and additional weaknesses may be discovered in the future. The countermeasure must be refined to secure it against these weaknesses, and continually re-examined in light of changes to network devices and protocols. Solutions to known weaknesses must be implemented and their effects on the effectiveness of the countermeasure studied.

2. Realism Improvements

Simulations using more realistic traffic sources and network topologies must be run to allow further validation and study of the DoS countermeasure. DiffServ client traffic was simulated by a CBR traffic source. This type of source is does not accurately model real-world network traffic, which tends to have greater variance in burst size and inter-burst timing. Simulations should be run using various traffic distributions, e.g.

exponential or Pareto distributions, to examine the performance of the countermeasure under more realistic traffic loads. Exponential and Pareto traffic generation applications are implemented in the simulator. However, they could not be evaluated due to synchronization effects observed at the DiffServ ingress router when using them. Additional study is required to tune these applications and eliminate synchronization.

3. Evaluation of Fairness

As discussed in Chapter IV, the fairness of the countermeasure with respect to individual flows has not been determined. A single traffic source was used to simulate client traffic. Follow-on work should be undertaken to create a simulation topology in which traffic from a number of different hosts is aggregated at the client gateway before exiting the client's domain. Modifications to the countermeasure should be developed to correct any inequities observed in the treatment of individual sub-flows of the aggregated traffic.

A topology for testing the fairness of the countermeasure can be implemented with only minor modifications to the simulator. The simulator provides implementations for protocols that are likely to be used for traffic requiring QoS guarantees. One example is the class *Agent/RTP*, which simulates the Real Time Protocol. A topology could be created in which a number of RTP agents in a client domain are provided QoS based on a single SLA between the client domain and the DiffServ domain. This would require all agents to send their traffic to a single client gateway, where the traffic would be aggregated and forwarded to the DiffServ network. Implementing the gateway requires creation of a new *Agent* subclass capable of performing these functions.

APPENDIX A. SOURCE CODE FOR DOS COUNTERMEASURE EXTENSIONS TO THE NS2 SIMULATOR

When the file *ns-allinone-2.1b8a.tar* is expanded, the main ns2 directory is created with the same name (without the *.tar* extension). Files listed in this appendix were added to the existing */ns2.1b8a/diffserv* subdirectory of the main ns2 directory, with the exception of *ns-diffserv.tcl*, which was added to the */ns2.1b8a/tcl/lib* directory.

attack-controller.h

```
#ifndef ns_attack_controller_h
#define ns_attack_controller_h

#include "udp.h"
#include "flood.h"

class FloodNode {
public:
   FloodNode();
   FloodAgent* agent;
   FloodNode* next;
};


class ControlAgent : public UdpAgent {
public:
  ControlAgent();
  ControlAgent(packet_t type);
  int command(int argc, const char*const* argv);
  void recv(Packet* p, Handler* h);
  inline ns_addr_t here() { return here_;}; //for use by sniffer agent

private:
   FloodNode* floodList;
};

#endif
```

attack-controller.cc

```
#include "attack-controller.h"
#include <iostream>
#include "ip.h"


//a linked list of the flood agents controlled by this node
FloodNode::FloodNode() {
  agent = NULL;
  next = NULL;
};


static class ControlClass : public TclClass {
public:
    ControlClass() : TclClass("Agent/UDP/Controller") {}
    TclObject* create(int, const char*const*) {
        return (new ControlAgent());
    }
} class_control_agent;


ControlAgent::ControlAgent() : UdpAgent() {
        floodList = NULL;
        bind("packetSize_", &size_);
}


ControlAgent::ControlAgent(packet_t type) : UdpAgent(type) {
        floodList = NULL;
        bind("packetSize_", &size_);
}


int ControlAgent::command(int argc, const char*const* argv) {
  Agent* srcAgent = NULL;

  if (argc == 3) {

    //implementation of tcl command "add-flooder"
    if (strcmp(argv[1], "add-flooder") == 0) {
      FloodAgent* newAgent = (FloodAgent*) TclObject::lookup(argv[2]);

      if (newAgent) {
        //in case feedback has already started
        newAgent->set_priority(prio_);
        FloodNode* newNode = new FloodNode();
        newNode->agent =  newAgent;
        newNode->next = floodList;
        floodList = newNode;
      } else {
        cout << argv[2] << " is not a valid flood agent " << endl;
      }
    return (TCL_OK);
    }
  }

  // If the command hasn't been processed by ControlAgent()::command,
  // call the command() function for the base class
  return (UdpAgent::command(argc, argv));
}


// received a sniffed code point
void ControlAgent::recv(Packet* p, Handler* h) {

  hdr_ip* iph = hdr_ip::access(p);

  // store the newly changed code pt as the controller's code pt
  prio_ = iph->prio();
  FloodNode* nodePtr = floodList;

  // Notify each flood agent of the CP change.  In reality this would
  // have to be transmitted over at least one link, which would take a
  // finite amount of time.  However, because the addresses of the
  // flood sources are all changed to match that of the target, sending
  // a packet to them breaks the Classifier code.  This hack takes zero
  // sim time, but we compensate by increasing the delay on the
  // flooder's outgoing links in the simulation script.
  while (nodePtr) {
    nodePtr->agent->set_priority(prio_);
    nodePtr = nodePtr->next;
  }

  // Discard the packet
  Packet::free(p);
}
```

dsFeedback.h

```
#include <map>
#include <deque>
#include "agent.h"
#include "ip.h"
#include "icmp.h"
#include "timer-handler.h"

#ifndef ds_feedback_h
#define ds_feedback_h
#define MAX_DROPS 3
#define DROP_WINDOW 5.0

using namespace std;

//specify STL template arguments and rename
typedef deque<double> timeDeque;
typedef timeDeque::iterator timeDequeIter;

class FeedbackInfo { //stores info about a specific src/dest pair
public:
   FeedbackInfo();
   timeDeque times;
   bool running;
   long cp1, cp2, seedKey;
   int maxDrops;
   double dropWin, keyWin;
};

typedef map<ns_addr_t, FeedbackInfo> dest2Fbi;
typedef dest2Fbi::iterator dest2FbiIter;
typedef pair<ns_addr_t,FeedbackInfo> destFbiPair;
typedef map<ns_addr_t,dest2Fbi> src2Dest;
typedef src2Dest::iterator src2DestIter;
typedef pair<ns_addr_t,dest2Fbi> srcDestPair;

class dsFeedback {
  friend class IcmpAgent;
public:
  static const int INVALID_CP;
  static void dropNotify(ns_addr_t src, ns_addr_t dest,
                         double dropTime, IcmpAgent* icmpAgent);
  static long generateNextCodePt(const long prev);
  static long generateInitCodePt(const long seedKey,
                                 const long secretKey);
  static bool isFeedbackRunning(ns_addr_t src, ns_addr_t dest);
  static bool isCodePtValid(ns_addr_t src, ns_addr_t dest,long codePt);
private:
  static src2Dest feedbackTable;
  static const long secretKey;
  static void sendFeedback (ns_addr_t src, IcmpAgent* icmpAgent,
                            long seed, double keyWindow);
};
#endif
```

dsFeedback.cc

```
#include "dsFeedback.h"
#include <iostream>
#include <math.h>

//initialize static members
src2Dest dsFeedback::feedbackTable;
const int dsFeedback::INVALID_CP = -1;
const long dsFeedback::secretKey = rand();

// define the less than operator for struct ns_addr_t (see config.h)
// so we can use it in STL containers
bool operator< (const ns_addr_t& n1, const ns_addr_t& n2) {
   return ( (n1.addr_ < n2.addr_) ||
   ((n1.addr_ == n2.addr_) && (n1.port_ < n2.port_)) );
}


/*********************************************************************
 * void dropNotify(src, dest, dropTime, icmpAgent)
 * PRE: None
 * POST: src/dest pair exists in feedbackTable with dropTime as the
 *       last entry in its timeDeque.  If MAX_DROPS is exceeded, a
 *       feedback message has been sent to src
 * RETURN: void
 *********************************************************************/
void dsFeedback::dropNotify(ns_addr_t src, ns_addr_t dest,
                            double dropTime, IcmpAgent* icmpAgent) {

  // check to see if this source has an entry already
  src2DestIter srcIter = feedbackTable.find(src);

  if (srcIter != feedbackTable.end()) {

    //if so, check to see if src is already paired with dest
    dest2FbiIter destIter = srcIter->second.find(dest);

    if (destIter != srcIter->second.end()) {

      //if so, add dropTime to the list
      FeedbackInfo* fbi = &(destIter->second);
      timeDeque* times_ = &(fbi->times);
      times_->push_back(dropTime);

      //remove all previous drop times outside the current window
      while (times_->front() < (dropTime - fbi->dropWin)) {
         times_->pop_front();
      }

      //if over drop limit for time window, and feedback not already
      //active for this pair, send feedback
      if ( !(fbi->running) && (times_->size() >= MAX_DROPS) ) {
        long seed = rand();
        double keyWindow = 0.100;  //100 milliseconds
        fbi->running = true;
        fbi->cp1 = 0;
        fbi->cp2 = generateInitCodePt(seed, secretKey);
        fbi->keyWin = keyWindow;
        fbi->seedKey = seed;
        icmpagent->sendFB(src, seed, keyWindow);
        return;
      }

    } else { //no entry for this (src,dest) pair
      FeedbackInfo fbi;
      srcIter->second.insert(destFbiPair(dest,fbi));

      // can't just insert dropTime, since one drop may be enough to
      // trigger feedback message.  Notify again instead
      dropNotify(src, dest, dropTime, icmpAgent);
    }

  } else {//add new entries for src,dest, and dropTime
    FeedbackInfo fbi;
    dest2Fbi d2f;
    d2f.insert(destFbiPair(dest,fbi));
    feedbackTable.insert(srcDestPair(src,d2f));

    // can't just insert dropTime, since one drop may be enough to
    // trigger feedback message.  Notify again instead
    dropNotify(src, dest, dropTime, icmpAgent);
  }
  return;
} //end dropNotify()


/*********************************************************************
 * bool isCodePtValid(src,dest,codePt)
 * PRE:  None
 * POST: if codePt = cp2 in the src/dest entry of the feedback table,
 *       cp1 is assigned cp2, and a new value for cp2 is generated.
 * RETURN: true if code point is valid or there is no entry for this
 *         src/dest pair, false otherwise
 ********************************************************************/
bool dsFeedback::isCodePtValid(ns_addr_t src, ns_addr_t dest,
                                                        long codePt) {

  src2DestIter srcIter = feedbackTable.find(src);

  if (srcIter != feedbackTable.end()) {
    dest2FbiIter destIter = srcIter->second.find(dest);

    if (destIter != srcIter->second.end()) {
      FeedbackInfo* fbi = &(destIter->second);

      if (fbi->running) { //don't check if FB isn't active

        if (fbi->cp1 != codePt) {

          if (fbi->cp2 != codePt) { return false; }
          else { // codePt = cp2
            fbi->cp1 = fbi->cp2;
            fbi->cp2 = generateNextCodePt(fbi->cp2);
          }
        }
      }
    }
  }
  return true;
} //end isCodePointValid()


/*********************************************************************
 * long generateNextCodePt(prev)
 * PRE: None
 * POST: None
 * RETURN: A code point between 1 and 255 generated based on the
 *         previous code point
 *********************************************************************/
long dsFeedback::generateNextCodePt(const long prev) {
  long nextCodePt;

  //replace this section with the desired hash function
  // hash function
  nextCodePt = prev % 255 + 1;
  // end hash function

  return nextCodePt;
} //end generateNextCodePt()


/*********************************************************************
 * bool feedbackRunning(src,dest)
 * PRE: None
 * POST: None
 * RETURN: True if the feedback mechanism is active for this src/dest
 *         pair, false otherwise
 *********************************************************************/
bool dsFeedback::isFeedbackRunning(ns_addr_t src, ns_addr_t dest) {
  src2DestIter srcIter = feedbackTable.find(src);

  if (srcIter != feedbackTable.end()) {
    dest2FbiIter destIter = srcIter->second.find(dest);

    if (destIter != srcIter->second.end()) {
      return destIter->second.running;

    } else { //in anticipation of checking this pair again later
      FeedbackInfo fbi;
      srcIter->second.insert(destFbiPair(dest,fbi));
    }

  } else { //in anticipation of checking this pair again later
    FeedbackInfo fbi;
    dest2Fbi d2f;
    d2f.insert(destFbiPair(dest,fbi));
    feedbackTable.insert(srcDestPair(src,d2f));
  }

  return false;
} //end feedbackRunning()


/*********************************************************************
 * int generateInitCodePt(const long seedKey, const long secretKey)
 * PRE: None
 * POST: None
 * RETURN: The initial code point based on seedKey and secretKey
 *********************************************************************/
long dsFeedback::generateInitCodePt(const long seedKey,
                                    const long secretKey) {
  long combinedKey = seedKey ^ secretKey;
  return generateNextCodePt(combinedKey);
} //end generateInitCodePt()


/*********************************************************************
 * FeedbackInfo()
 * Constructor for class FeedbackInfo
 *********************************************************************/
FeedbackInfo::FeedbackInfo() {
  running = false;
  cp1 = 0;
  cp2 = 0;
  maxDrops = MAX_DROPS;
  dropWin = DROP_WINDOW;
  keyWin = 0;
  seedKey = 0;
}
```

icmp.h

```
#ifndef ns_icmp_h
#define ns_icmp_h

#include "agent.h"
#include "tclcl.h"
#include "packet.h"
#include "ip.h"
#include "sniffer.h"


struct hdr_icmp {
  ns_addr_t origSrc;
  long seed;        // seed value for the codePt hash function
  double codePtWin; // how often the sender should change code points

  static int offset_;
  inline static int& offset() { return offset_; }
  inline static hdr_icmp* access(const Packet* p) {
    return (hdr_icmp*) p->access(offset_);
  }
};


class CodePtTimer : public TimerHandler {
public:
   CodePtTimer (SnifferAgent* srcAgent, long newCP, double codePtWin);
   void expire(Event* e);

protected:
   SnifferAgent* agent;
   long currentCP;
   double delay;
};


class IcmpAgent : public Agent {
public:
  IcmpAgent();
  IcmpAgent(packet_t type);
  int command(int argc, const char*const* argv);
  void recv(Packet*, Handler*);
  void sendFB(ns_addr_t src, long seed, double keyWindow);

private:
  CodePtTimer* timer;
};
#endif
```

icmp.cc

```
#include "icmp.h"
#include "dsFeedback.h"

//initialize static member
int hdr_icmp::offset_;


// CodePtTimer constructor
CodePtTimer::CodePtTimer(SnifferAgent* src, long newCP, double cpWin)
       : TimerHandler(), agent(src), delay(cpWin), currentCP(newCP){};

// Definition of virtual function declared in base class TimerHandler
void CodePtTimer::expire(Event* e) {

   if (agent != 0) {
      cout << "Agent changing code pt..."  <<  endl;
      currentCP = dsFeedback::generateNextCodePt(currentCP);
      agent->set_priority(currentCP);
      resched(delay);
   }
}

// Add ICMP header to OTcl hierarchy
static class IcmpHeaderClass : public PacketHeaderClass {
public:
      IcmpHeaderClass() : PacketHeaderClass("PacketHeader/ICMP",
                                    sizeof(hdr_icmp)) {
            bind_offset(&hdr_icmp::offset_);
        }
      void export_offsets() {
            field_offset("origSrc", OFFSET(hdr_icmp, origSrc));
            field_offset("seed", OFFSET(hdr_icmp, seed));
            field_offset("codePtWin", OFFSET(hdr_icmp, codePtWin));
      }
} class_icmphdr;


// Add ICMPAgent class to OTcl hierarchy
static class IcmpClass : public TclClass {
public:
      IcmpClass() : TclClass("Agent/ICMP") {}
      TclObject* create(int, const char*const*) {
            return (new IcmpAgent());
      }
} class_icmp;

// IcmpAgent constructors
IcmpAgent::IcmpAgent() : Agent(PT_ICMP) {
      bind("packetSize_", &size_);
}


IcmpAgent::IcmpAgent(packet_t type) : Agent(type) {
      bind("packetSize_", &size_);
}


int IcmpAgent::command(int argc, const char*const* argv) {

  if (argc == 2) {

    if (strcmp(argv[1], "send") == 0) {
      Packet* p = allocpkt();
      hdr_icmp* hdr = hdr_icmp::access(p);
      hdr->seed = 2;
      hdr->codePtWin = 0.005;
      send(p,0);
    }
  }

  // If the command hasn't been processed by IcmpAgent()::command,
  // call the command() function for the base class
  return (Agent::command(argc, argv));
}


/*********************************************************************
 * void recv(pkt,h)
 * PRE:  Packet pkt contains the port number of the agent whose
 *       dropped packet caused the feedback message to be sent
 * POST: DoS countermeasure is started for that agent
 * RETURN: void
 *********************************************************************/
void IcmpAgent::recv(Packet* pkt, Handler* h) {
  hdr_icmp* hdr = hdr_icmp::access(pkt);
  long seed = hdr->seed;

  // Call the OTcl procedure 'Agent/Icmp get-agent-handle {port}'
  // which returns a handle to the source agent (see ns-diffserv.tcl)
  char out[128];
  sprintf(out, "%s get-agent-handle %d", name(), hdr->origSrc.port_);
  Tcl& tcl = Tcl::instance();
  tcl.eval(out);
  SnifferAgent* src = (SnifferAgent*) TclObject::lookup(tcl.result());

  if (src != 0) {
    //Calculate new code point and change it
    long newCP =
          dsFeedback::generateInitCodePt(seed, dsFeedback::secretKey);
    src->set_priority(newCP);
    //create timer and schedule the next code point change
    timer = new CodePtTimer(src, newCP, hdr->codePtWin);
    timer->sched(hdr->codePtWin);
  } else {
    cout << "No object in lookup table for " << tcl.result() << endl;
  }

  // free the memory assigned to the received packet
  Packet::free(pkt);
}
```

```
/*********************************************************************
 * void recv(pkt,h)
 * PRE:  None
 * POST: Feedback message sent to src
 * RETURN: void
 *********************************************************************/
void IcmpAgent::sendFB(ns_addr_t src, long seed, double keyWindow) {
  Packet* p = allocpkt();
  hdr_icmp* hdr = hdr_icmp::access(p);
  hdr->origSrc = src;
  hdr->seed = seed;
  hdr->codePtWin = keyWindow;
  send(p,0);
}
```

ns-diffserv.tcl

```
Agent instproc set_priority {newprio} {
    $self instvar prio_
    set prio_ newprio
}


Agent instproc prio {} {
    $self instvar prio_
    return $prio_
}


Agent/ICMP instproc get-agent-handle {port} {
    $self instvar node_
    set src_agent [$node_ agent $port]
    return [$node_ agent $port]
}
```

sniffer.h

```
#ifndef ns_sniffer_h
#define ns_sniffer_h

#include "udp.h"
#include "attack-controller.h"


class SnifferAgent : public UdpAgent {
public:
  SnifferAgent();
  SnifferAgent(packet_t type);
  inline int prio() { return prio_;};

  // override functions in parent classes
  int command(int argc, const char*const* argv);
  void sendmsg(int nbytes, const char* flags);
  void set_priority(int priority);

private:
  bool codePtChanged;
  ControlAgent* controller;
  //ControlAgent defined in attack-controller.{h,cc}
};

#endif
```

sniffer.cc

```
#include "sniffer.h"
#include <iostream>
#include "ip.h"

// Add SnifferAgent to thew OTcl hierarchy
static class SnifferClass : public TclClass {
public:
      SnifferClass() : TclClass("Agent/UDP/Sniffer") {}
      TclObject* create(int, const char*const*) {
            return (new SnifferAgent());
      }
} class_sniffer_agent;

// SnifferAgent constructors
SnifferAgent::SnifferAgent() : UdpAgent() {
        controller = NULL;
        codePtChanged = true;
      bind("packetSize_", &size_);
}

SnifferAgent::SnifferAgent(packet_t type) : UdpAgent(type) {
        controller = NULL;
        codePtChanged = true;
      bind("packetSize_", &size_);
}


int SnifferAgent::command(int argc, const char*const* argv) {
  Agent* srcAgent = NULL;

  if (argc == 3) {

    // C++ implmentation of OTcl
    // 'Agent/UDP/Sniffer set-controller <agent-handle>'
    if (strcmp(argv[1], "set-controller") == 0) {
      ControlAgent* newAg = (ControlAgent*)TclObject::lookup(argv[2]);

      if (newAg != NULL) {
        controller = newAg;
      } else {
        cout << argv[2] <<  " is not a valid agent " << endl;
      }
    return (TCL_OK);
    }
  }

  // If the command hasn't been processed by SnifferAgent()::command,
  // call the command() function for the base class
  return (UdpAgent::command(argc, argv));
}


void SnifferAgent::sendmsg(int nbytes, const char* flags) {
  Packet* snfPkt = NULL;

  // notify controller only when code point has changed
  if (codePtChanged && (controller != NULL)) {
    codePtChanged = false;
    snfPkt = allocpkt();
    hdr_ip* ip_snf = hdr_ip::access(snfPkt);
    ip_snf->dst_ = controller->here();
  }

  //need to ensure that packet is sent before it is "sniffed"
  UdpAgent::sendmsg(nbytes, flags);

  if (snfPkt != NULL) {
    // send "sniffed" code point to controller
    Connector::send(snfPkt,0);
  }
}


void SnifferAgent::set_priority(int priority) {
  prio_ = priority;
  // ensure that a changed code point triggers a message to the
  // sniffer controller when the next packet is sent by this source
  codePtChanged = true;
}
```

flood.h

```
#ifndef ns_flood_h
#define ns_flood_h

#include "udp.h"

class FloodAgent : public UdpAgent {
public:
  FloodAgent();
  FloodAgent(packet_t type);
  int command(int argc, const char*const* argv);
  void recv(Packet* p, Handler* h);
  inline ns_addr_t here() { return here_;};
};

#endif
```

flood.cc

```
#include "flood.h"
#include "ip.h"

static class FloodClass : public TclClass {
public:
      FloodClass() : TclClass("Agent/UDP/Flood") {}
      TclObject* create(int, const char*const*) {
            return (new FloodAgent());
      }
} class_flood_agent;


FloodAgent::FloodAgent() : UdpAgent() {
      bind("packetSize_", &size_);
}


FloodAgent::FloodAgent(packet_t type) : UdpAgent(type) {
      bind("packetSize_", &size_);
}


int FloodAgent::command(int argc, const char*const* argv) {
  // call the command() function for the base class
  return (UdpAgent::command(argc, argv));
}

void FloodAgent::recv(Packet* p, Handler* h) {
  hdr_ip* iph = hdr_ip::access(p);
  set_priority(iph->prio());

  // Discard the packet
  Packet::free(p);
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. MODIFIED NS2 SOURCE CODE

Files listed in this appendix are modified versions of existing ns2 files. Modifications are labeled [FEEDBACK]. For brevity, unmodified sections have been omitted.

agent.h

```
/*
 * Copyright (c) 1993-1997 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgement:
 *    This product includes software developed by the Computer Systems
 *    Engineering Group at Lawrence Berkeley Laboratory.
 * 4. Neither the name of the University nor of the Laboratory may be
 *    used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS''
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
*/
#ifndef ns_agent_h
#define ns_agent_h

#include "connector.h"
#include "packet.h"
#include "timer-handler.h"
#include "ns-process.h"
#include "app.h"
```

```
#define TIMER_IDLE 0
#define TIMER_PENDING 1

/*
 * Note that timers are now implemented using timer-handler.{cc,h}
 */

#define TRACEVAR_MAXVALUELENGTH 128

class Application;


// store old value of traced vars
// work only for TracedVarTcl
struct OldValue {
    TracedVar *var_;
    char val_[TRACEVAR_MAXVALUELENGTH];
    struct OldValue *next_;
};

class Agent : public Connector {
 public:
    Agent(packet_t pktType);
    virtual ~Agent();
    void recv(Packet*, Handler*);

    //added for edrop tracing - ratul
    void recvOnly(Packet *) {};

    void send(Packet* p, Handler* h) { target_->recv(p, h); }
    virtual void timeout(int tno);

    virtual void sendmsg(int sz, AppData*, const char* flags = 0);
    virtual void send(int sz, AppData *data) { sendmsg(sz, data, 0); }
    virtual void sendto(int sz, AppData*, const char* flags = 0);

    virtual void sendmsg(int nbytes, const char *flags = 0);
    virtual void send(int nbytes) { sendmsg(nbytes); }
    virtual void sendto(int nbytes, const char* flags, nsaddr_t dst);
    virtual void connect(nsaddr_t dst);
    virtual void close();
    virtual void listen();
    virtual void attachApp(Application* app);
    virtual int& size() { return size_; }
    inline nsaddr_t& addr() { return here_.addr_; }
    inline nsaddr_t& port() { return here_.port_; }
    inline nsaddr_t& daddr() { return dst_.addr_; }
    inline nsaddr_t& dport() { return dst_.port_; }
    void set_pkttype(packet_t pkttype) { type_ = pkttype; }
    void set_priority(int priority) { prio_ = priority; } //[FEEDBACK]

 protected:
    int command(int argc, const char*const* argv);
    virtual void delay_bind_init_all();
    virtual int delay_bind_dispatch(const char *varName,
                             const char *localName, TclObject *tracer);
    virtual void recvBytes(int bytes);
    virtual void idle();
    Packet* allocpkt() const;  // alloc + set up new pkt
    Packet* allocpkt(int) const;    // same, but w/data buffer
    void initpkt(Packet*) const;    // set up fields in a pkt

    ns_addr_t here_;          // address of this agent
    ns_addr_t dst_;                 // destination address for pkt flow
    int size_;                // fixed packet size
    packet_t type_;                 // type to place in packet header
    int fid_;                 // for IPv6 flow id field
    int prio_;                // for IPv6 prio field
    int flags_;               // for experiments (see ip.h)
    int defttl_;              // default ttl for outgoing pkts

#ifdef notdef
    int seqno_;         /* current seqno */
    int class_;         /* class to place in packet header */
#endif

    static int uidcnt_;

    Tcl_Channel channel_;
    char *traceName_;         // name used in agent traces
    OldValue *oldValueList_;

    Application *app_;        // ptr to application for callback

    virtual void trace(TracedVar *v);
    void deleteAgentTrace();
    void addAgentTrace(const char *name);
    void monitorAgentTrace();
    OldValue* lookupOldValue(TracedVar *v);
    void insertOldValue(TracedVar *v, const char *value);
    void dumpTracedVars();
 private:
    void flushAVar(TracedVar *v);
};

#endif
```

dsred.h

```
/* Copyrights (c) 2000 Nortel Networks
**********************************************************************
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 * 3. All advertising materials mentioning features or use of this
 *    software
 *    must display the following acknowledgement:
 *      This product includes software developed by Nortel Networks.
 * 4. The name of the Nortel Networks may not be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY NORTEL AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL NORTEL OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
 * OF SUCH DAMAGE.
 *
 * Developed by: Farhan Shallwani, Jeremy Ethridge
 *               Peter Pieda, and Mandeep Baines
 * Maintainer: Peter Pieda <ppieda@nortelnetworks.com>
 */

/*
 * dsred.h
 *
 * The Positions of dsREDQueue, edgeQueue, and coreQueue in the Object
 * Hierarchy.
 *
 * This class, i.e. "dsREDQueue", is positioned in the class hierarchy
 * as follows:
 *
 *             Queue
 *               |
 *           dsREDQueue
 *
 *
 *   This class stands for "Differentiated Services RED Queue".  Since
 * the original RED does not support multiple parameters, and other
 * functionality needed by a RED gateway in a Diffserv architecture,
 * this class was created to support the desired functionality.  This
 * class is then inherited by two more classes, moulding the old
 * hierarchy as follows:
 *
 *
 *             Queue
 *               |
 *           dsREDQueue
 *           |        |
 *     edgeQueue    coreQueue
 *
 *
 * These child classes correspond to the "edge" and "core" routers in a
 * Diffserv architecture.
 *
 */


#ifndef dsred_h
#define dsred_h

#include "red.h"  // need RED class specs (edp definition, for example)
#include "queue.h"      // need Queue class specs
#include "dsredq.h"
#include "agent.h"
#include "icmp.h" //[FEEDBACK]


/* The dsRED class supports the creation of up to MAX_QUEUES physical
 * queues at each network device, with up to MAX_PREC virtual queues in
 * each queue. */
#define MAX_QUEUES 8 // maximum number of physical RED queues
#define MAX_PREC 3  // maximum number of virtual RED queues in one
                    // physical queue
#define MAX_CP 256      // maximum number of code points in a
simulation
#define MEAN_PKT_SIZE 1000    // default mean packet size, in bytes,
                              // needed for RED calculations

enum schedModeType {schedModeRR, schedModeWRR, schedModeWIRR,
schedModePRI};

#define PKT_MARKED 3
#define PKT_EDROPPED 2
#define PKT_ENQUEUED 1
#define PKT_DROPPED 0


/*-----------------------------------------------------------------------
struct phbParam
    This struct is used to maintain entries for the PHB parameter
table, used
to map a code point to a physical queue-virtual queue pair.
----------------------------------------------------------------------*/
struct phbParam {
   int codePt_;
   int queue_;    // physical queue
   int prec_;     // virtual queue (drop precedence)
};

struct statType {
      long drops;       // per queue stats
      long edrops;
      long pkts;
      long valid_CP[MAX_CP];  // per CP stats
      long drops_FB; // [FEEDBACK]packets droped due to feedback
      long drops_CP[MAX_CP];
      long edrops_CP[MAX_CP];
      long pkts_CP[MAX_CP];
};


/*--------------------------------------------------------------------
class dsREDQueue
    This class specifies the characteristics for a Diffserv RED router.
--------------------------------------------------------------------*/

class dsREDQueue : public Queue {
public:
   dsREDQueue();
   int command(int argc, const char*const* argv); // interface to ns
                                                  //scripts

protected:
  redQueue redq_[MAX_QUEUES]; // the physical queues at the router
  NsObject* de_drop_;         // drop_early target
  IcmpAgent* icmpAgent; //[FEEDBACK] the icmp agent associated with
                        // this queue
  statType stats; // used for statistics gatherings
  int qToDq; // current queue to be dequeued in a round robin manner
  int numQueues_; // the number of physical queues at the router
  int numPrec;   // the number of virtual queues in each physical queue
  phbParam phb_[MAX_CP];            // PHB table
  int phbEntries;     // the current number of entries in the PHB table
  int ecn_;          // used for ECN (Explicit Congestion Notification)
  LinkDelay* link_;           // outgoing link
    int schedMode;                  // the Queue Scheduling mode
    int queueWeight[MAX_QUEUES];    // A queue weight per queue
    double queueMaxRate[MAX_QUEUES]; // Max Rate for Priority Queueing
    double queueAvgRate[MAX_QUEUES]; // Avg Rate for Priority Queueing
    double queueArrTime[MAX_QUEUES]; // Arr Time for Priority Queueing
      int slicecount[MAX_QUEUES];
      int pktcount[MAX_QUEUES];
      int wirrTemp[MAX_QUEUES];
      unsigned char wirrqDone[MAX_QUEUES];
  int queuesDone;

  void reset();
  void edrop(Packet* p); // used so flowmonitor can monitor early drops
  void enque(Packet *pkt); // enques a packet
  Packet *deque(void);  // deques a packet
  int getCodePt(Packet *p); // given a packet, extract the code point
```

```
                            // marking from its header field
  void selectQueueToDeque();// round robin scheduling dequing algorithm
  void lookupPHBTable(int codePt, int* queue, int* prec); // looks up
                 //queue and prec numbers corresponding to a code point
  void addPHBEntry(int codePt, int queue, int prec); // edits phb entry
                                                     // in the table
  void setNumPrec(int curPrec);
  void setMREDMode(const char* mode, const char* queue);
  void printStats(); // print various stats
  double getStat(int argc, const char*const* argv);
  void printPHBTable();  // print the PHB table
  void setSchedularMode(const char* schedtype); //Sets the schedular
                                                // mode
  void addQueueWeights(int queueNum, int weight); // Add a maxRate to a
                                                  // PRI queue
  void addQueueRate(int queueNum, int rate); // Add a weigth to a WRR
                                             // or WIRR queue
  void printWRRcount();       // print various stats
  void applyTSWMeter(Packet *pkt); // apply meter to calculate average
                                   // rate of a PRI queue
};

#endif
```

dsred.cc

```
/* Copyrights (c) 2000 Nortel Networks
**********************************************************************
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 * 3. All advertising materials mentioning features or use of this
 *    software
 *    must display the following acknowledgement:
 *      This product includes software developed by Nortel Networks.
 * 4. The name of the Nortel Networks may not be used
 *    to endorse or promote products derived from this software without
 *    specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY NORTEL AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL NORTEL OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF
 * USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
 * ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
 * OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
 * OF SUCH DAMAGE.
 *
 * Developed by: Farhan Shallwani, Jeremy Ethridge
 *               Peter Pieda, and Mandeep Baines
 * Maintainer: Peter Pieda <ppieda@nortelnetworks.com>
 */

#include <stdio.h>
#include "ip.h"
#include "dsred.h"
#include "delay.h"
#include "random.h"
#include "flags.h"
#include "tcp.h"
#include "dsredq.h"

//[FEEDBACK]
#ifdef ds_feedback_h
  #include "dsFeedback.h"
#endif
```

.
.
.

```
/*---------------------------------------------------------------------
---------
void enque(Packet* pkt)
    The following method outlines the enquing mechanism for a Diffserv
router.
This method is not used by the inheriting classes; it only serves as an
outline.
-----------------------------------------------------------------------
-------*/
void dsREDQueue::enque(Packet* pkt) {
  int codePt, queue, prec;
  hdr_ip* iph = hdr_ip::access(pkt);
  ns_addr_t src = iph->src();
  ns_addr_t dst = iph->dst();
  bool dropped = false; //[FEEDBACK]

  codePt = iph->prio();//extracting the marking done by the edge router
  int ecn = 0;
  double now = Scheduler::instance().clock();

  stats.pkts++;

// [FEEDBACK]
#ifdef ds_feedback_h
  //if codePt is invalid, drop without any other processing
  if (codePt == dsFeedback::INVALID_CP) {
    cout << "Dropped due to invalid CP" << endl;
    stats.drops_FB++; // [FEEDBACK] increment count of
                      // feedback related drops
    stats.drops++;
    drop(pkt);
    dropped = true;
  }
#endif

  if (!dropped) {  //[FEEDBACK] allow further processing
    //looking up queue and prec numbers for that codept
    lookupPHBTable(codePt, &queue, &prec);

    // code added for ECN support
    hdr_flags* hf = hdr_flags::access(pkt);
//line 200
    if (ecn_ && hf->ect()) ecn = 1;

    stats.pkts_CP[codePt]++;

    switch(redq_[queue].enque(pkt, prec, ecn)) {
      case PKT_ENQUEUED:
        break;

      case PKT_DROPPED:
        stats.drops_CP[codePt]++;
        stats.drops++;
        cout << "Dropped by dsredq." << endl;

#ifdef ds_feedback_h
        //[FEEDBACK] notify sender of packet dropped by edge router
        if (!dsFeedback::isFeedbackRunning(src,dst)) {
          dsFeedback::dropNotify(src, dst, now, icmpAgent);
        }
#endif

        drop(pkt);
        break;

      case PKT_EDROPPED:
        stats.edrops_CP[codePt]++;
        stats.edrops++;
        cout << "Early dropped by dsredq." << endl;

#ifdef ds_feedback_h
        //[FEEDBACK] notify sender of packet dropped by edge router
        if (!dsFeedback::isFeedbackRunning(src,dst)) {
          dsFeedback::dropNotify(src, dst, now, icmpAgent);
        }
#endif

        edrop(pkt);
        break;

      case PKT_MARKED:
        hf->ce() = 1;   // mark Congestion Experienced bit
        break;

      default:
        break;
    }
  }
}

                                    .
                                    .
                                    .

/*----------------------------------------------------------------------
---------
int command(int argc, const char*const* argv)
    Commands from the ns file are interpreted through this interface.
----------------------------------------------------------------------
-------*/
int dsREDQueue::command(int argc, const char*const* argv)
{

                                    .
                                    .
                                    .


  if (argc == 3) {  //[FEEDBACK] set the icmp agent handle

    if (strcmp(argv[1], "set-icmp-agent") == 0) {
      Tcl& tcl = Tcl::instance();
      IcmpAgent* agent = (IcmpAgent*) TclObject::lookup(argv[2]);

      if (agent == NULL) {
        tcl.resultf("[dsRED][FEEDBACK] No agent %s", argv[2]);
      return (TCL_ERROR);
      }

      icmpAgent = agent;
      return(TCL_OK);
    }
  }


  return(Queue::command(argc, argv));
}
```

APPENDIX C. SAMPLE OTCL SCRIPT

test_default.tcl

```
#----------------------------------------------------------------------
# test_default.tcl
# Author: Matthew Braun
# Dates: Aug 28, 2002
# Notes: Heavily modified from the example file
# ~ns/tcl/ex/diffserv/ds-cbr-TSW3CM.tcl written by Jeremy Ethridge.
#
#  ----       ------            ------   ----   ------   ----   ------
#  |s1|-------|ext1|------------|ext2|---|e1|---|core|---|e2|---|dest|
#  ----       ------            ------   ----   ------   ----   ------
#                |                |
#                |                |
#                |                |
#                |                |
#                |                |
#                |                |
#             --------          ----
#             |attack|----------|z1|
#             --------          ----
#
#----------------------------------------------------------------------

#    W = 2d
#  r+A = 2CIR
# good source sends 2500 packets (2.0 seconds) while under attack

set ns [new Simulator]
set nf [open test_def.nam w]
set dest_trace [open dest.trace w]
#$ns namtrace-all $nf

proc finish {} {
     global ns nf
     $ns flush-trace
     close $nf
#     exec nam test_def.nam &
     exit 0
}

# difference between paths
  set delta "110ms"

# tokenBucket parameters
  set cir0   10000000
  set cbs0       4001

# source parameters rates
  set rate0  10000000
  set rate1  10000000
  set packetSize 1000
```

```
# pareto (or exponential) parameters
  set burstTime 500ms
  set idleTime  20ms
  set parShape 1.9

# port numbers
  set udpPort 1
  set icmpPort 2
  set cbrPort 1

# traffic type in use
#  set traf_type "Exponential"
#  set traf_type "Pareto"
 set traf_type "CBR"

# simulation parameters
  set testTime 2.5

# Set up the network topology shown at the top of this file:
$ns node-config -addressingType heir
set source1 [$ns node]

set attacker [$ns node]
set extern1 [$ns node]
set extern2 [$ns node]

# Set src addr of malicioius packets to the addr of the good src
set flood1 [$ns node [$source1 node-addr]]
set flood2 [$ns node [$source1 node-addr]]

set edge1 [$ns node]
set core [$ns node]
set edge2 [$ns node]
set dest [$ns node]

#create links between nodes
$ns duplex-link $source1 $extern1 100Mb 10ms DropTail
#set link_src_ext [$ns link $source1 $extern1]
#$source1 add-route [$attacker id] [$link_src_ext head]

$ns duplex-link $extern1 $attacker 100Mb $delta DropTail
$ns duplex-link $extern1 $extern2  100Mb 100ms DropTail
$ns duplex-link $attacker $zombie1 1Mb 100ms DropTail
$ns duplex-link $attacker $zombie2 1Mb 100ms DropTail
$ns duplex-link $zombie1 $extern2  1000Mb  100ms DropTail
$ns duplex-link $zombie2 $extern2  1000Mb  100ms DropTail
$ns duplex-link $extern2 $edge1    1000Mb  10ms DropTail


$ns duplex-link $edge2 $dest 100Mb 5ms DropTail
$ns simplex-link $edge1 $core 100Mb 5ms dsRED/edge
$ns simplex-link $core $edge1 100Mb 5ms dsRED/core
$ns simplex-link $core $edge2 100Mb 5ms dsRED/core
$ns simplex-link $edge2 $core 100Mb 5ms dsRED/edge
```

```
#nam layout of nodes
$ns duplex-link-op $source1 $extern1 orient down-right
$ns duplex-link-op $extern1 $extern2 orient right
$ns duplex-link-op $attacker $extern1 orient up
$ns duplex-link-op $attacker $zombie1 orient right
$ns duplex-link-op $attacker $zombie2 orient right
$ns duplex-link-op $zombie1 $extern2 orient up
$ns duplex-link-op $zombie2 $extern2 orient up
$ns duplex-link-op $extern2 $edge1 orient right
$ns duplex-link-op $edge1 $core orient right
$ns duplex-link-op $core $edge2 orient right
$ns duplex-link-op $edge2 $dest orient down


#create dsred queues on the simplex links in the DS domain
set qE1C [[$ns link $edge1 $core] queue]
set qE2C [[$ns link $edge2 $core] queue]
set qCE1 [[$ns link $core $edge1] queue]
set qCE2 [[$ns link $core $edge2] queue]


# Set DS RED parameters from Edge1 to Core:
$qE1C meanPktSize $packetSize
$qE1C set numQueues_ 2
$qE1C setNumPrec 1
$qE1C setMREDMode DROP
$qE1C setSchedularMode PRI
$qE1C addQueueRate 0 $cir0
$qE1C addPolicyEntry [$source1 node-addr]
                        [$dest node-addr] TokenBucket 10 $cir0 $cbs0
$qE1C addPolicerEntry TokenBucket 10 0
$qE1C addPHBEntry  0 1 0
$qE1C addPHBEntry 10 0 0
$qE1C configQ 0 0 10 20 0.99
$qE1C configQ 1 0  0  0 1.00


# Set DS RED parameters from Core to Edge2:
$qCE2 meanPktSize $packetSize
$qCE2 set numQueues_ 2
$qCE2 setNumPrec 1
$qCE2 addPHBEntry  0 1 0
$qCE2 addPHBEntry 10 0 0
$qCE2 configQ 1 0  0  0 1.00
$qCE2 configQ 0 0 10 20 0.10


# Set up one connection between good source and the destination:
set udp1 [new Agent/UDP/Sniffer]
$udp1 set packetSize_ $packetSize
$udp1 set prio_ 0
$source1 attach $udp1 $udpPort
set valid_traffic [new Application/Traffic/$traf_type]
$valid_traffic attach-agent $udp1
$valid_traffic set packet_size_ $packetSize
$valid_traffic set rate_ $rate0
$valid_traffic set random_ 1
```

```
# $valid_traffic set burst_time_ $burstTime
# $valid_traffic set idle_time_ $idleTime
# $valid_traffic set shape_ $parShape


#null sink for traffic to dest
set null1 [new Agent/Null]
$dest attach $null1 $udpPort
$ns connect $udp1 $null1


# Set up control agent at attacker
set attk1 [new Agent/UDP/Controller]
$attk1 set packetSize_ $packetSize
$attk1 set prio_ 0
$attacker attach $attk1

#connect this controller to the sniffer on source1
$udp1 set-controller $attk1

# create zombie agents
set f1 [new Agent/UDP/Flood]
set f2 [new Agent/UDP/Flood]


#create cbr sources to attach to zombies
set cbr1 [new Application/Traffic/CBR]
$cbr1 attach-agent $f1
$cbr1 set packet_size_ $packetSize
$cbr1 set rate_ $rate1
$cbr1 set random_ 1
set cbr2 [new Application/Traffic/CBR]
$cbr2 attach-agent $f2
$cbr2 set packet_size_ $packetSize
$cbr2 set rate_ $rate1
$cbr2 set random_ 1

#attach flood agents to nodes
$flood1 attach $f1 $udpPort
$flood2 attach $f2 $udpPort

#connect zombie agents to dest agent
#set null2 [new Agent/Null]
#$dest attach $null2 5
$ns connect $z1 $null1
$ns connect $z2 $null1

#add flooders to controller
$attk1 add-flooder $z1
$attk1 add-flooder $z2

$udp1 set class_ 1
$f1 set class_ 2
$f2 set class_ 3

$ns color 1 Green
$ns color 2 Red
```

```
$ns color 3 Black

# Set up icmp agents at the good source node and the DS Edge
set icmp1 [new Agent/ICMP]
$source1 attach $icmp1 $icmpPort

set icmp2 [new Agent/ICMP]
$edge1 attach $icmp2 $icmpPort

#need to let this queue know what it's attached icmp agent is
#so it can tell the agent to send feedback msgs
$qE1C set-icmp-agent $icmp2
$ns connect $icmp1 $icmp2

#tracing and mointoring objects
set dropt [$ns create-trace Drop $dest_trace $source1 $dest]
$ns drop-trace $edge1 $core $dropt

$qE1C printPolicyTable
$qE1C printPolicerTable

$ns at 0.1 "$valid_traffic start"
$ns at 0.5 "$qE1C printStats"
$ns at 0.5001 "$cbr1 start"
$ns at 0.5003 "$cbr2 start"
$ns at 1.50 "$qE1C printStats"

$ns at $testTime "$valid_traffic stop"
$ns at $testTime "$cbr1 stop"
$ns at $testTime "$cbr2 stop"
$ns at [expr $testTime + 0.2] "$qE1C printStats"
$ns at [expr $testTime + 0.21] "finish"

$ns run
```

THIS PAGE INTENTIONALLY LEFT BLANK

REFERENCES

[BRAUN02] M. Braun and G. Xie. “A Feedback Mechanism for Mitigating Denial of Service Attacks against Differentiated Services Clients,” paper presented at the 10th International Conference on Telecommunications Systems, Modeling, and Analysis. Monterey, California. 4 October 2002.

[FALL02] K. Fall and K. Varadhan, eds. The ns Manual. February 2002. Available at *http://www.isi.edu/nsnam/ns/ns-documentation.html.*

[LEE01] H. Lee and K. Park. On The Effectiveness of Probabilistic Packet Marking for IP Traceback Under Denial of Service Attack. *Proceedings of IEEE INFOCOM 2001*, Anchorage, Alaska. April 2001.

[NS02] “The Network Simulator – ns2.” [http://www.isi.edu/nsnam/ns/]. September 2002.

[PARK01] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. *Proceedings of ACM SIGCOMM 2001 Conference,* pp. 15-26, Zurich, Switzerland, September 2001.

[RFC791] J. Postel, ed, Internet Protocol. RFC 791, IETF, September 1981.

[RFC864] J. Postel. Character Generator Protocol. RFC 864, IETF, May 1983.

[RFC1810] J. Touch. Report on MD5 Performance. RFC 1810, IETF, June 1995.

[RFC2402] S. Kent and R. Atkinson. IP Authentication Header. RFC 2402, IETF, November 1998.

[RFC2474] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, IETF, December 1998.

[RFC2475] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang and W. Weiss. An Architecture for Differentiated Services, RFC 2475, IETF, December 1998.

[RFC2827] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which Employ IP Source Address Spoofing. RFC 2827, May 2000.

[RIPEMD96] H. Dobbertin, A. Bosselaers, B. Preneel, ``RIPEMD-160, a strengthened version of RIPEMD," *Fast Software Encryption*, *LNCS 1039*, D. Gollmann, Ed., Springer-Verlag, 1996, pp. 71-82.

[SAVAGE00] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. *Proceedings of ACM SIGCOMM 2000 Conference*, pp. 295-306, Stockholm, Sweden, August 2000.

[YAU01] D. Yau, F. Liang, and J. Lui. On Defending against Distributed Denial-of-service Attacks with Server-centric router throttles. Technical Report, CERIAS and Department of Computer Science, Purdue University. May 2001.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Prof. Geoffrey Xie, Code CS/Xg
   Naval Postgraduate School
   Monterey, California

4. Lieutenant Matthew Braun
   1938 N. Fairfield
   Chicago, IL, 60647