Hop-Count Filtering: Simulation and Analysis

By

Elie Salem

A Thesis

Submitted in Partial Fulfillment of the Requirements for

the Degree of Master of Science in

Computer Information System

Department of Computer Science

Faculty of Natural and Applied Sciences

Notre Dame University – Louaize

Zouk Mosbeh, Lebanon

Fall 2006

Hop-Count Filtering: Simulation and Analysis

By
Elie Salem

Committee Members:

Dr. Hikmat Farhat: Assistant Professor of Computer Science, Advisor and Chairperson

Dr. Hoda Maalouf: Assistant Professor of Computer Science.

Dr. Marie Khair: Associate Professor of Computer Science.

Dr. Ziad Rached: Assistant Professor of Mathematics.

Date of Thesis Defense: April 12, 2006

Acknowledgment

First, I would like to thank my advisor Dr. Hikmat Farhat for his consistent support throughout my studies and research. Second, I am thankful for the love and support expressed by my parents and my brother. Their constant support has carried me through some difficult challenges. Finally, I appreciate the help I have received from my friends... who have given of themselves and their time to help me in achieving my thesis study.

Abstract

A denial of service (DoS) attack is an attempt by a person or a group of persons to cripple an online service. This can have serious consequences on companies like Amazon and eBay, which rely on their online availability to do business. Many defense mechanisms have been proposed to thwart DoS attacks. One of these defense mechanisms is called hop-count filtering. Although an attacker can forge any field in the IP header, he or she cannot falsify the number of hops an IP packet takes to reach its destination. This hop-count information can be inferred from the time-to-live value in the IP header. Using a mapping between IP addresses and their hop-counts to an Internet server, the hop-count filtering can distinguish spoofed IP packets from legitimate ones. In this research, we will simulate the hop-count filtering using SSFNet. And then we analyze the results to show the efficiency of this technique in protecting against DoS attacks.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

DDoS	Distributed denial of service
DML	Domain modeling language
DoS	Denial of service
DRDoS	Distributed reflection denial of service
HCF	Hop-count filtering
ICMP	Internet Control Message Protocol
IP	Internet protocol
IP2HC	IP-to-hop-count
IRC	Internet relay chat
LAN	Local area network
MULTOPS	Multi-level tree for online packet statistics
NAT	Network address translation
NIC	Network interface card
SSFNet	Scalable simulation framework network
TCP	Transmission control protocol
TTL	Time to live
UDP	User Datagram Protocol

Chapter 1

Introduction and Problem Definition

1.1 Introduction

A denial of service attack is an attempt by a person or a group of persons to cripple an online service. This can have serious consequences, especially for companies like Amazon and eBay, which rely on their online availability to do business. Recently, there have been some large-scale attacks targeting high-profile Internet sites. Consequently, a good deal of effort is now being made to come up with mechanisms to detect and diminish such attacks. In general, the attacks can be of three forms:

- Attacks that exploit some vulnerability in the software implementation of a service.
- Attacks that use up all the available resources at the victim machine.
- Attacks that consume all the bandwidth available to the victim machine. These are called "bandwidth attacks."

A distributed framework becomes especially suited for such attacks since a reasonable amount of data directed from a number of hosts can generate a lot of traffic at and near the target machine, clogging all the routes to the victim. Protection against such large-scale distributed bandwidth attacks is one of the most difficult and urgent problems facing today's Internet. Community Emergency Response Team (CERT) reports that bandwidth attacks have become the most common form of denial of service attacks.

1.2 Problem Definition

As one of the most difficult problems in network security, distributed denial of service attacks, also known as DDoS, have posed a serious threat to the availability of Internet services. Instead of subverting services, DDoS attacks limit and block legitimate

users' access by exhausting the victim server's resources, or saturating stub networks' access links to the Internet. Attackers often spoof internet protocol (IP) addresses by randomizing the 32-bit source-address field in the IP header in order to conceal flooding sources and localities in flooding traffic [3]. Moreover, some known DDoS attacks, such as Smurf and more recent distributed reflection denial of service attacks (DRDoS), are not possible without IP spoofing. Such attacks pretend to be the source IP address of each spoofed packet with the victim's IP address.

1.3 Feasibility of DDoS Attacks

The Internet was designed with functionality in mind, not security. The TCP/IP protocol, which is the most widely used protocol for data communication, assumes that all the hosts participating in the communication have no malicious intent. There is no security built into the Internet infrastructure to protect hosts from other hosts not regulating their own behavior. For example, the TCP protocol assumes that hosts will reduce the rate of packet transmission on detecting packet losses due to congestion. If, instead, a particular host does not respond to the congestion conditions, it can easily overwhelm the intermediate links to the destination. Such a design opens up the Internet to many opportunities for denial of service attacks [1]. Some of the Internet features that make DoS attacks possible are:

- **Weakness of Internet Security.** DDoS attacks are launched from hosts whose security has been weakened. No matter how secure a particular host is, it opens itself to the possibility of a DDoS attack if there are other insecure hosts in the Internet which can be used to launch such attacks.

- **Difficulty in Tracing Back the Attack to the Source.** Most of the Internet runs on top of the TCP/IP protocol. The underlying protocol, IP, is connectionless in nature. At each intermediate step from the source to the destination, the decision is made about the next host to forward the packet. All such routing decisions are made on the basis of the destination address. It is thus possible to generate packets with incorrect source IP addresses and use them to launch denial of service at-

tacks. This technique is known as IP spoofing. Users with sufficient privileges on a system have the ability to construct such fake packets. For example, in Linux, raw sockets can be created which enable users (with super-user access) to construct all the packet contents and headers for a given packet. This renders the task of determining the true source of attack very difficult. Apart from the source IP address, attackers nowadays randomly change all the headers in an IP datagram, keeping only the destination address constant. This makes dropping packets based on certain characteristics very difficult, because distinguishing attack packets from legitimate packets becomes hard. Some attacks also rely on illegitimate source addresses to launch a denial of service attack on the hosts whose IP address was used—i.e., the Smurf attack. If, upon detection of an attack, packets are dropped exclusively on the basis of the IP source addresses, the hosts whose IP addresses were used for the spoofing will suffer a denial of service.

- **Limited Resources.** The infrastructure of the interconnected hosts and networks is composed of limited resources. Bandwidth processing power and storage capacities are all targets of denial of service attacks. Increasing these resources substantially just makes it harder for an attack to be effective. Even if the attack is not able to shut down the victim completely, it may waste its resources, reducing the level of quality as seen by the end users, and making the service provider incur heavy financial losses.

- **A Target-Rich Environment.** There are thousands and thousands of hosts and networks in the Internet with vulnerabilities that can be exploited. It is therefore easy to gain control of many of these hosts that can then be used to launch DDoS attacks.

- **Easier to Break Systems than to Make Them.** It is easier to break the networking infrastructure protocols than to develop them since, at the time of the software design, no one foresaw the system being used for malicious purposes. This can lead to unexpected behavior of the network systems as a response to unexpected

packets. For instance, all routers and hosts allocate buffers in memory while waiting for all the fragments constituting a datagram to arrive. If such a router receives badly formed fragments indicating a very large datagram, it will keep on allocating huge amounts of memory until it runs out of memory and cannot process more datagrams.

1.4 Research Objectives

The objective of this research is to simulate the hop-count filtering scheme using SSFNet. We will then analyze the results to show its efficiency in protecting against DDoS attacks.

1.5 Thesis Organization

The rest of the paper is organized as follows. Chapter 2 is a brief overview of denial of service attacks along with their classifications. Chapter 3 discusses the defense mechanisms used to prevent DDoS. Chapter 4 describes in detail the hop-count filtering technique which is used in the simulation. Chapter 5 explains the tools and features of the simulation along with the results analysis. Finally, chapter 6 addresses what the future might hold with respect to DDoS attacks.

Chapter 2

Background and Motivation

2.1 Overview: TCP Connections

Individual TCP packets contain "flag bits" which specify the contents and purpose of each packet. For example, a packet with the "SYN" (synchronize) flag bit set initiates a connection from the sender to the recipient. A packet with the "ACK" (acknowledge) flag bit set acknowledges the receipt of information from the sender. A packet with the "FIN" (finish) flag bit set terminates the connection from the sender to the recipient.

The establishment of a TCP connection typically requires the exchange of three Internet packets between two machines in an interchange known as TCP three-way hand-shake. This is shown in Figure 1 below:

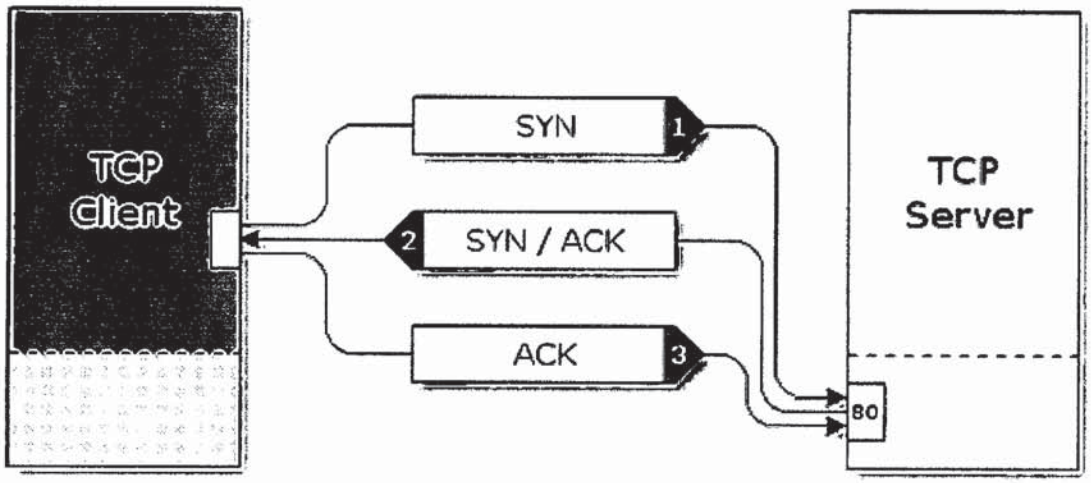

Figure 1: Legitimate TCP connection.

A TCP client (such as a Web browser, ftp client, etc.) initiates a connection with a TCP server by sending a SYN packet to the server. When a connection-requesting SYN packet is received, the server's operating system replies with a connection accepting the SYN/ACK packet. Finally, when the client receives the server's acknowledging

SYN/ACK packet for the pending connection, it replies with an ACK packet and the two-way TCP connection is established between the client and the server.

2.2 TCP Exploit: The Traditional SYN Flood

Several years ago, a weakness in the TCP connection handling of many operating systems was discovered and exploited by malicious Internet hackers. As shown in the TCP transaction diagram in Figure 2, the server's receipt of a client's SYN packet causes the server to prepare for a connection. It typically allocates memory buffers for sending and receiving the connection's data. Then it records the various details of the client's connection including the client's remote IP and connection port number. In this way, the server will be prepared to accept the client's final connection-opening ACK packet. Also, if the client's ACK packet should fail to arrive, the server will be able to resend its SYN/ACK packet, presuming that it might have been lost or dropped by an intermediate Internet router. This means that memory and other significant server resources are allocated as a consequence of the receipt of a single Internet SYN packet. Clever but malicious Internet hackers figured that there had to be a limit to the number of "half-open" connections a TCP server could handle, and they came up with a simple means for exceeding those limits.

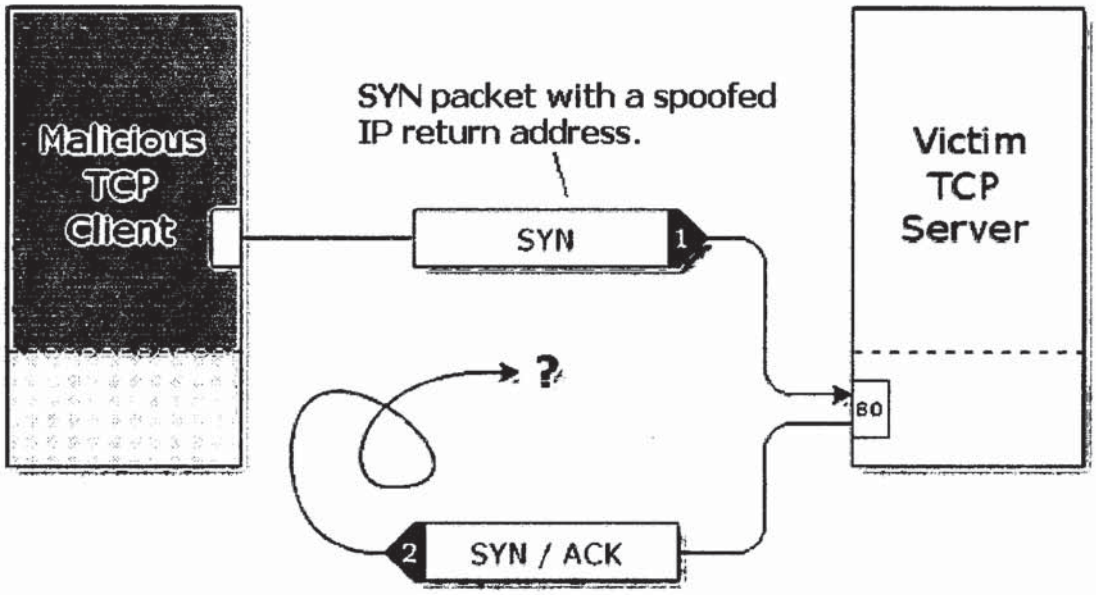

Figure 2: Spoofed TCP connection.

Through the use of "raw sockets," the packet's "return address" (source IP) can be overridden and falsified. When a SYN packet with a spoofed source IP arrives at the server, it appears as any other valid connection request. The server will allocate the required memory buffers, record the information about the new connection, and send an answering SYN/ACK packet back to the client.

But since the source IP contained in the SYN packet was deliberately falsified (it is often a random number), the SYN/ACK will be sent to a random IP address on the Internet. If the packet were addressed to a valid IP, the machine at that address might reply with a "RST" (reset) packet to inform the server that it did not request a connection. But with over 4 billion Internet addresses, the chances are that there will be no machine at the address and the packet will be discarded.

The problem is that the server has no way of knowing that the malicious client's connection request was fraudulent, thus treating it like any other valid pending connection. It needs to wait for some time for the client to complete the three-way handshake. If the ACK is not received, the server needs to resend the SYN/ACK in the belief that it might have been lost on its way back to the client. All of this connection management consumes valuable and limited resources in the server. Meanwhile, the attacking TCP client continues firing fraudulent SYN packets at the server, forcing it to accumulate a continuously growing pool of incomplete connections. At some point, the server will be unable to accommodate any more half-open connections and even valid connections will fail, since the server's ability to accept *any* connections will have been maliciously consumed.

This DoS attack was not "distributed." A single, malicious, SYN-generating machine, hiding its Internet address and identity behind falsified source IP SYN packets, could tie up and bring down a large Web site. Because of the stateless and destination-based routing of the Internet, it is difficult to counter IP spoofing. The IP protocol cannot prevent a sender from hiding the origin of its packets. In addition, destination-based routing does not maintain state information on senders and forwards each IP packet toward its destination without validating the packet's true origin. Overall, IP spoofing makes DDoS attacks much more difficult to defend against.

2.3 DDOS Classification

The different types of denial of service attacks can be broadly classified into software exploits and flooding attacks. Flooding attacks can be further classified into single- and multi-source attacks based on the number of attackers. This classification is depicted in Figure 3 and explained next.

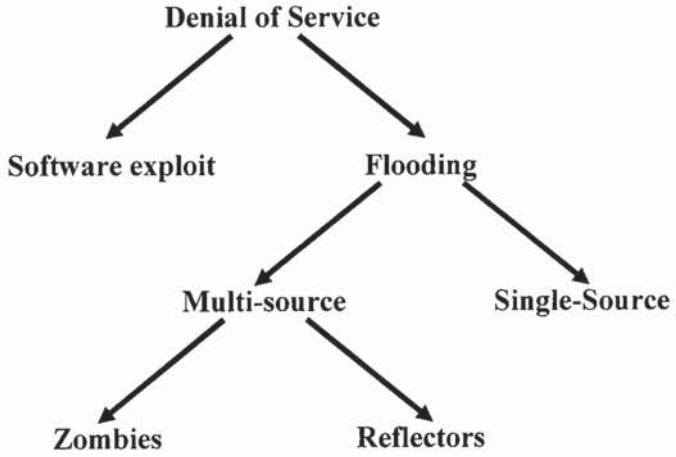

Figure 3: Classification of DoS attacks.

2.3.1 Software Exploits

These attacks exploit specific software bugs in the operating system or an application, and can potentially disable the victim machine with a single or a few packets. A well-known example is the "ping of death," which causes the operating system to crash by sending a single large ICMP echo packet. Similarly, the "land attack" sends a single TCP SYN packet containing the victim's IP address in both the source and destination address fields, resulting in an endless loop in the protocol stack. Such attacks can only be prevented by applying software updates.

2.3.2 Flooding Attacks

Flooding attacks are the result of one or more attackers sending constant streams of packets designed to overwhelm the victim's link bandwidth or computing resources. Flooding attacks are classified as single-source attacks when a single zombie is flooding the victim and multi-source when multiple zombies are flooding the victim, as shown in Figures 4 and 5, respectively. Multiple attackers may organize for an attack in order to

increase firepower or to evade detection. In both types, the attacker can install attack tools on the host machine that can generate illegal packets. Examples of such attacks are the TCP NULL attack that generates packets with no flags set, the Xmas attack that has all TCP flags set, and attacks that use packets with a nonexistent IP protocol number. Several attack tools are available on the Internet, such as Trinoo [2], Stacheldraht [2], Mstream [2], and Tribal Flood Network 2000 [2], which generate flooding attacks with a combination of TCP, UDP, and ICMP packets [11].

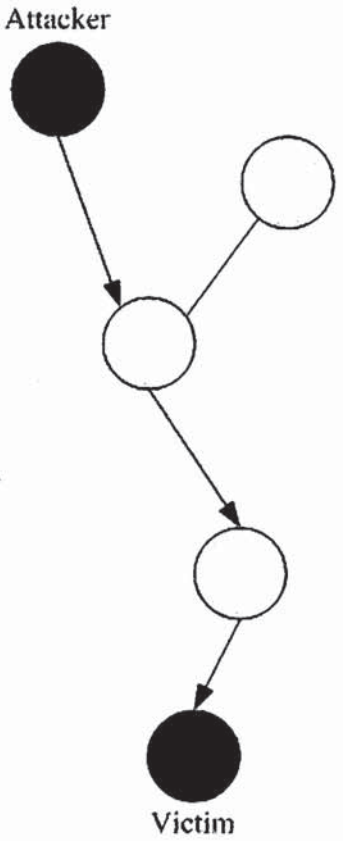

Figure 4: Single-source attack.

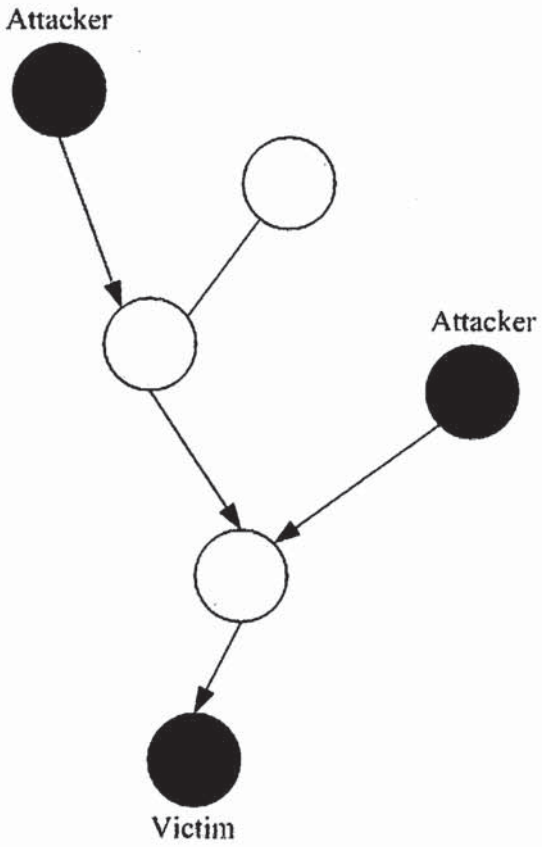

Figure 5: Multi-source attack.

A significant percentage of captured attacks consist of a single source. Moore et al. determined that 14% of all DoS attacks were directed toward home machines using either dial-up or broadband access [12]. Most DoS attacks on the Internet are from a single source to a single victim. Thus, a single high-bandwidth zombie can generate enough packets to overwhelm a victim.

The third type of attack is the reflector attack shown in Figure 6. Such attacks are used to hide the identity of the attacker and/or to amplify an attack. A reflector is any host that responds to requests, such as Web servers or ftp servers. Examples of such requests are TCP SYN requests responding with TCP SYN-ACK packets or ICMP echo requests responding with ICMP echo replies. Servers may be used as reflectors by spoofing the victim's IP address in the source field of the request, tricking the reflector into directing its response to the victim. Unlike direct zombie attacks, reflector attacks require well-formed packets to solicit a reply. If many reflector machines are used, such an attack can

easily overwhelm the victim without adversely affecting the reflectors. Reflectors can also be used as amplifiers by sending packets to the broadcast address on the reflector network, requesting a response from every host on the local area network (LAN). Reflector attacks may be more difficult to prevent because reflectors are often hosts intentionally providing Internet services. For instance, a Web site like Yahoo could be used as a reflector to launch DDoS attacks, although it is providing normal services (i.e., search engine and e-mail) for Internet users.

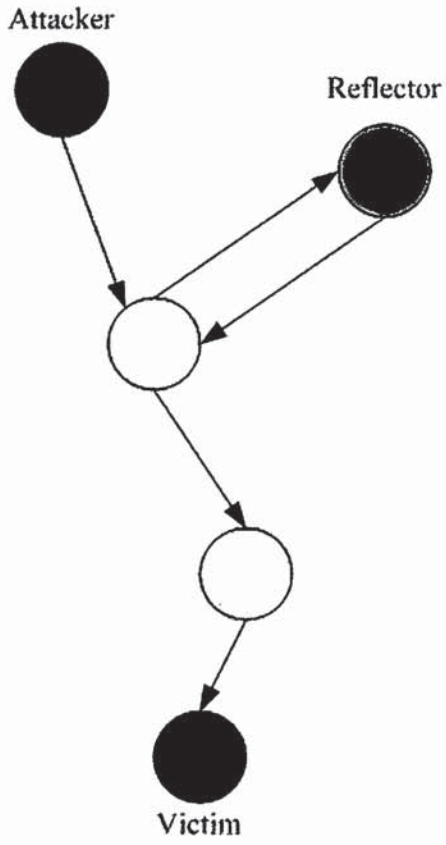

Figure 6: Reflector attack.

2.4 Bandwidth Attacks

Figure 7, below, helps clarify the consequences of a bandwidth attack. The computers and/or networks shown to the right are serviced by the central "aggregation router." This router is placed at the "customer edge" of the Internet service provider's network to collect and disperse traffic from many smaller customer networks. Thus,

many lower bandwidth Internet connections are aggregated into a single high-bandwidth Internet connection for routing to the public Internet.

During normal operation, the traffic coming from the Internet down the "big pipe" will be sorted and forwarded to the router's various lower bandwidth client networks. But when the big pipe is filled by a high volume of packets bound for just one of the client networks, the router will find itself faced with the task of squeezing too many packets from the big pipe into the much smaller pipe. It has no choice but to deliberately drop and discard a large percentage of the packets struggling to get through the smaller pipe. Valid Internet clients, trying to access the resources on the far side of the smaller pipe, will re-send their dropped packets, but these clients will generally give up after a few attempts. The victim's network is effectively blasted off the Internet by the flood of malicious traffic.

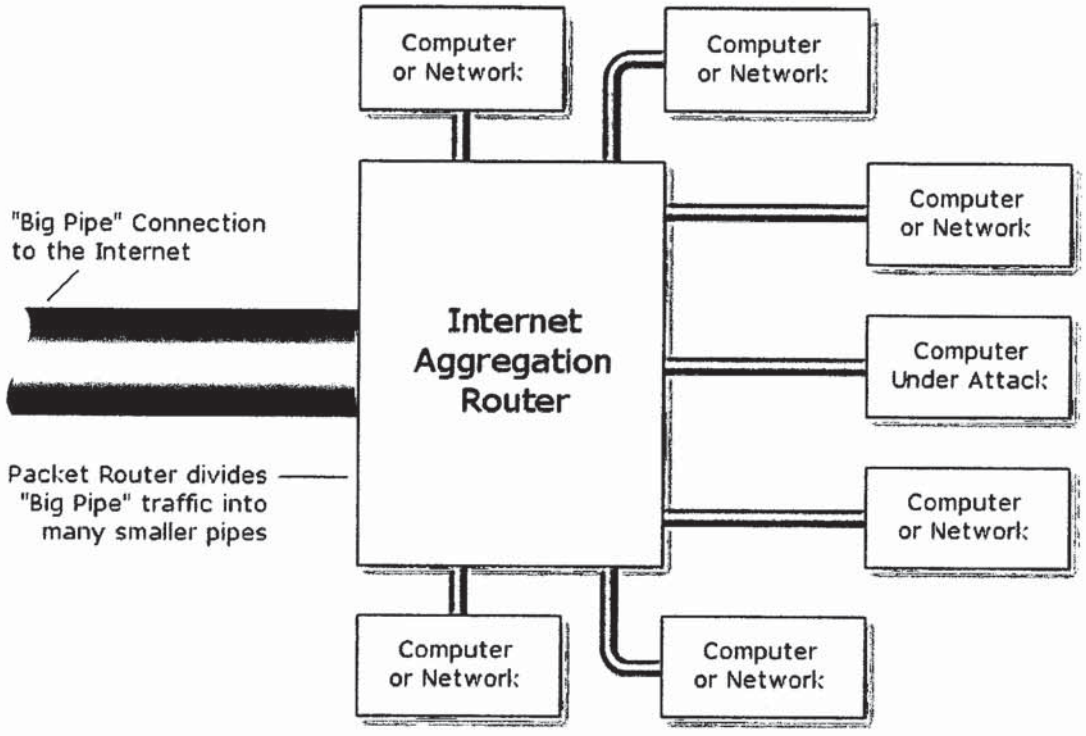

Figure 7: Discarding packets.

2.5 Progress of DDoS Attacks

DDoS attacks gained widespread notoriety and media exposure with the three days of DoS attacks (February 7–11, 2000) that were launched against major Internet sites like CNN, Yahoo, EBay, and Datek. Multiple attack tools like Trinoo [2], TFN [2], StachleDraht [2], and TFN2K [2] were used; the sophistication of the DDoS attack tools has improved with time. Therefore, a historical study of DDoS attacks also gives a good overview of the various techniques that are used in organizing such attacks.

2.5.1 Steps in Organizing a DDoS Attack

A DDoS attack is carried out by a group of machines (referred to as "agents" or "zombies") that start sending packets to a victim host on receiving commands from a machine (referred to as a "handler" or a "master") under the control of the attacker. Therefore, an attacker has to take the following steps in order to launch a DDoS attack [8].

a) Compromising Hosts: Compromise a number of hosts and deploy software on them that converts them into agents or handlers in the attack network: An attacker first scans the network, looking for vulnerable hosts. The next step is to try and exploit a known vulnerability in order to compromise the system. After compromising the system, the attacker deploys the software that converts the system into an agent or a handler. The attacker then leaves after trying to cover its tracks. Some tools carry out one more step called the "propagation step," in which all the compromised hosts recursively start the whole process of scanning, exploitation, and deployment. In the early days of DDoS technology development, all the four steps (scanning, compromising, deployment, and propagation) would be done manually by the attacker. Over time, more and more automation has been brought into each of these steps. The T0rnkit toolkit, essentially a rootkit with some DoS attack capabilities, was probably the first DDoS toolkit to automate the process of scanning, exploitation, and deployment. Manual intervention was needed to carry out propagation. With the advent of Internet worms, starting with the Ramen worm, the propagation step has also been automated. Internet worms are a malicious piece of software that automatically scan and compromise a host, after which they deploy copies of themselves in the compromised

host. Therefore, the number of hosts that are compromised by them increases exponentially. These worms can then be used to launch denial of service attacks, while at the same time carrying out the scanning to detect other vulnerable hosts. Sometimes, these worms generate such high packet rates just due to aggressive scanning, which itself leads to disruption in the normal operation of the networks being scanned.

b) Method of Propagation: DDoS attack tools are using increasingly sophisticated methods of self-propagation. The evolution in propagation methodologies has gone through the following steps.

- Central chain propagation: After the compromise of a host, the mechanism used to carry out the compromise copies the attack toolkit from a central server to the new machine. Transfer protocols like HTTP, FTP, or RCP are used in this transfer. In this method, discovery of the central server disables further propagation.

- Back-chaining propagation: The attack toolkit is copied from the attacking host onto the compromised host.

- Autonomous propagation: Some worms like the Morris worm of 1988 and the Code Red worm hard code the attack instructions into the code itself. This makes the propagation extremely fast, and no external file transfer has to be initiated.

c) Establish the Communication Channels between the Attacker, Handlers, and Agents: As the DDoS network starts building, it becomes hard for the attacker to keep track of all the agents that can be signaled to start a DDoS attack. The handlers facilitate this task by communicating with the agents. Therefore, most agents will listen to handlers for commands at some well-known ports. Handlers, in turn, need to keep track of all the agents under their control, which they can do by maintaining the list of all agents in local files. The need for agents and handlers to communicate also makes it easier to detect the presence of DDoS tools in a network. For example, if it is known that Trinoo [2] agents listen to handlers at the UDP port 31335, the existence of an open port

with the same number can alert the system administrators to the presence of the Trinoo attack tool in the network. A DDoS network can also be disabled by simply filtering out packets to that particular port. Similarly, a system integrity checker like Tripwire [2] can signal the presence of suspicious files that may be used to maintain agent lists. On getting ahold of a handler's agent list, all the agents can be identified and disabled. As a result, there has been a constant evolution and increased sophistication in the techniques used to avoid detection of agent/handler communication. These include using encryption and IRC channels as the communication backbone for the DDoS network. IRC-driven DDoS networks are sometimes called "botnets" because they are software-driven participants rather than human participants. Agents can then connect to well-known IRC servers and be issued commands by the attacker or the handlers which also first connect to the same IRC servers. Since it is hard to distinguish these connections to the server from other legitimate connections, it becomes harder to detect the DDoS networks. Identification of an agent or a handler will take one no further than identifying a couple of IRC servers being used by this DDoS network. To make matters worse, recent DDoS tools also come with the ability to send configuration commands from handlers to agents, which change the IRC server or channel being used. Therefore, the DDoS network can hop from server to server, making detection even harder.

d) Launching a DDoS Attack: Once the DDoS network has been set up and the infrastructure for communication between agents and handlers established, all that an attacker needs to do is to issue commands to the agents to start sending packets to the victim host. The agents try to send unusual data packets (i.e., all TCP flags set, repeated TCP SYN packets, large ICMP packets) to maximize the possibility of disrupting the victim and the intermediate nodes. Certain basic packet attack types are preferred by the attack tool designers. All the attack tools use a combination of these packet attack types to launch a DDoS attack. The basic attack types are:

- TCP floods: Streams of packets with various flags (SYN, RST, ACK) are sent to the victim machine. The TCP SYN flood works by exhausting the TCP connection queue of the host and thus denying legitimate connection requests. TCP ACK

floods can disrupt the nodes corresponding to the host addresses of the floods. Also, the tool that uses TCP ACK flooding, known as "mstream," has been known to cause disruptions in a router even with a moderate packet rate. Both TCP SYN flooding and the mstream attack constitute a group of attacks called "asymmetric attacks," in which a less powerful system can render a much more powerful system useless. An interesting variation of the TCP flood is a flashcrowd—when many users simultaneously try to access a popular Web site, making it temporarily unavailable.

- ICMP floods (i.e., ping floods): A stream of ICMP packets is sent to the victim host. A variant of the ICMP floods is the Smurf attack in which a spoofed IP packet consisting of an ICMP ECHO_REQUEST is sent to a directed broadcast address. The reference for ICMP specifies that no ECHO_REPLY packets should be generated for broadcast addresses, but unfortunately many operating systems and router vendors have failed to implement. As a result, the victim host receives ICMP ECHO_REPLY packets from all the hosts on the network, increasing the chance of a crash. Such networks are known as "amplifier networks," thousands of which have been documented.

- UDP floods: A huge amount of UDP packets are sent to the victim host. Trinoo is a popular DDoS tool that uses UDP floods as one of its attack payloads.

Over time, the evolution of DDoS tools has mainly been in the increased sophistication of scanning, exploitation, propagation, and agent/handler communication. The attack traffic which is generated has remained largely the same (i.e., TCP, ICMP, UDP floods) because there is no real need to evolve the attack payload since the existing ones are very effective.

The complexity and scale of DDoS attacks have continued to increase with time and are expected to do so in the future as well. Consequently, there is an urgent need to come up with DDoS detection and mitigation techniques. These DDoS defense mechanism are explained in chapter 3.

Chapter 3

Defense against DDoS Attacks

3.1 Introduction

DDoS attacks pose the most potent threat to the network infrastructure today. Sadly, no truly effective mechanisms are in place to defend against an ongoing DDoS attack. There has been active research in this field, but most of the solutions proposed are either in the initial stages of development or assume more functionality from the Internet protocol, which is difficult to put into effect.

Since DDoS attack mitigation causes such a challenge, more pressure should be placed on preventing such attacks. This would require a more conscious effort in the security of an organization and its internal networks. The first step that any organization should take is to come up with a security policy, with part of it dedicated to DDoS attack prevention and mitigation. There should be explicit mention of the steps to be taken before, during, and after a DDoS attack.

- **Before the attack:** The first step to prevent a DDoS attack is to avoid compromise and the use of hosts as agents. To achieve this, the whole network should be guarded by a firewall. The time between the posting of an exploit and its use by attackers is continuously diminishing. System and network administrators need to keep up with this pace and diligently apply patches supplied by the various vendors. This very important aspect should have explicit mention in the security policy of any company. Responsibilities for keeping track of various patches should be divided among a group of people on the basis of operating system and major software applications. This brings up the related issue of the level of training and motivation of system administrators. Organizations should consider spending more on the training and compensation of system administrators. They should be able to efficiently and quickly detect the onset of a DDoS attack and the type of

attack tool used in order to aid law agencies in tracking down the hosts belonging to the DDoS network and possibly the attacker himself.

- **During the attack:** During a DoS attack, it becomes difficult for system administrators to get access to the routers and servers of their network. So there should be some understanding between the organization and its upstream service provider, which is in a better position to throttle down the attack packets and gather information for aiding in forensic analysis. Furthermore, with some re-engineering, the whole network should be built in such a way that separate network operations are isolated from the underlying data transfer network. This network could be composed of various routers which can be used to turn off packet rates and maybe transfer data in order to be used later on in forensic analysis.

- **After the attack:** It is absolutely imperative to have an intrusion response team in place. This team should be able to gather data after an attack in order to identify the type of attack being carried out. Such analysis can aid in tracking down the hosts that form the DDoS network so that they can be shut down. There should also be closer cooperation between law enforcement agencies in order to gather and submit evidence that can be used to prosecute an attacker.

3.2 Defense Mechanisms

To prevent DDoS attacks, researchers have taken two distinct approaches: router-based and victim-based. The router-based approach makes improvements to the routing infrastructure, whereas the victim-based approach enhances the resilience of Internet servers against attacks.

The router-based approach performs either off-line analysis of flooding traffic or on-line filtering of DDoS traffic inside routers. Off-line IP traceback attempts to establish procedures to track down flooding sources after occurrences of DDoS attacks. Although it does help pinpoint locations of flooding sources, off-line IP traceback does not help sustain service availability during an attack. On-line filtering mechanisms rely on IP router enhancements to detect abnormal traffic patterns and thwart DDoS attacks. How-

ever, these solutions require not only router support but coordination among different routers and networks, and widespread deployment.

The victim-based approach has the advantage of being immediately deployable, compared with the router-based approach. Furthermore, a potential victim has a much stronger incentive to deploy defense mechanisms than do network service providers. The current victim-based approach protects Internet servers using sophisticated resource management schemes—for example, by shielding interactive video traffic from bulk data transfers. However, without a mechanism to detect and discard spoofed traffic, spoofed packets will share the same resource principals and code paths as legitimate requests. While a resource manager can confine the scope of damage to the service under attack, he may not be able to sustain the availability of the service. However, the server's ability to filter most, if not all, spoofed IP packets can help sustain service availability even under DDoS attacks.

3.2.1 Ingress/Egress Filtering [5, 6]

Ingress/egress filtering makes it difficult for attackers to launch attacks using spoofed IP addresses. IP spoofing is required for some attacks like the Smurf (ping flood) to work. Furthermore, IP spoofing makes it difficult to trace back the attack to the actual originating host. If upon detection of a DDoS attack, the traffic is dropped based on just the IP source address, the network whose source address was spoofed is also denied access. This, in itself, is a denial of service for the end-users on that network. Any firewall connecting a network to the Internet will have some interfaces connected to the internal network and some connected to the Internet. The firewall should apply ingress filtering on the external interfaces and drop all packets that have the source address that belongs to its internal network, since such packets have been clearly spoofed. If such packets are allowed into the network, the attacker can pretend to be a host within the same network. There is generally a higher level of trust between hosts on the same network, which can lead to security compromises. Egress filtering is applied on the internal interface on packets that are heading out of the network. The firewall drops all the packets that have source addresses that do not belong to the local network. This stops an attacker from using hosts within that network as DDoS agents. If these two solutions are widely deployed

all over the Internet, they will go a long way toward stopping attacks that rely on IP spoofing to be effective. Furthermore, they will enable easy traceback of the attacks to the true origin, since the attacking hosts are forced to use their true IP addresses. Ingress/egress filtering does not provide protection against bandwidth-based DDoS attacks. This type of filtering depends on widespread use to be effective. Unfortunately, few ISPs today enforce ingress filtering, either due to lack of awareness, the administrative burden, or to allow applications like Mobile IP to work.

There are, however, some proposals [9] that enhance the routing protocol on the Internet to automatically drop packets that may have spoofed IP source addresses. These protocols can be enhanced to incorporate information about valid source-address prefixes that can be observed on a particular router interface. The router's forwarding function normally just observes the destination IP address on a packet and decides the next hop to which to forward the packet. This function can now be enhanced to drop packets that have been determined to have invalid prefixes based on the routing information exchange.

3.2.2 IP Traceback

IP traceback is the process of tracing back the forged IP packet to its true source, rather than the spoofed IP address that was used in the attack. At a very basic level, one can think of this as a manual process in which the administrator of the network under attack places a call to his ISP (whose router is just one hop away), asking for the direction from which the attack packets are coming. The upstream ISP determines that information, but has to call its own upstream router to find out the previous hop, and so on. Therefore, cooperation between different networks is required in order to trace back attack packets to their true sources. Since the manual traceback is very tedious, there have been various proposals recently to automate this process. The three main ways of doing a packet traceback are given below.

Link-Testing Schemes: In this scheme, the victim tests each of its incoming links as a probable input link for the DDoS traffic. Burch and Cheswick [14] propose a scheme called "controlled flooding" to determine the loaded link. The victim generates some load on each of the links coming into it and observes the perturbation in the input packet rate.

The idea is that the loaded link will suffer from the most perturbation. This process is recursively followed as the victim generates loads across links further upstream until the source is reached. This scheme assumes that the victim has the complete map of all the paths to all its possible Internet sources. Furthermore, this analysis can only be done during a DDoS attack. However, it would be difficult for the victim to generate the packets for flooding while under a DDoS attack. Some people have stated that controlled flooding of various links might in itself constitute a denial of service attack. Link-testing mechanisms work best when there is a single attacking source but give bad results under a DDoS attack.

Packet-Marking Schemes: In packet-marking schemes, each router in addition to forwarding a packet, inserts a mark in the packet. The mark is a unique identifier corresponding to this particular router. As a result, the victim can determine all the intermediate hops for each packet by observing the inserted marks. There are two variants to this marking scheme. First is the deterministic packet-marking scheme in which each router marks all the packets passing through it with its unique identifier. This scheme is thus similar to the IP record-route option. This makes the reconstruction of the attack path at the victim trivial. But the downside to this scheme is that routers decelerate as they have to perform additional functionality. Furthermore, the packet headers can grow up to an arbitrary size, and so provisions must be made for this. The attacker can then generate bogus packets that already have false information filled in the limited number of entries available. To overcome the deficiencies of the deterministic packet-marking scheme, a probabilistic packet-marking scheme has been proposed, in which there is just a single entry in the IP header to store the markings. Each router on the path from the source to the destination writes down its unique identifier in the entry in the packet header with some probability. By writing in the entry, routers overwrite any previous entry that had been present. The victim can reconstruct the path from the source to itself upon receiving a large number of packets. It is even possible to reconstruct the order of routers from the source to the victim based on the relative frequencies of the router markings in the packets. If a packet follows the path A → B → C → D → E → F, where B, C, D, and E are intermediate routers, the relative frequencies of the packet markings found at F would be in the decreasing sequence E → D → C →B, since E would overwrite the markings made

by the previous routers. A downside of this scheme is that some packets will not be overwritten by any of the routers. The attacker can therefore write bogus information in all the packets, knowing that some of these packets will get through and confuse the victim [10]. This method also does not work well for denial of service attacks that can work without a lot of packets, because it requires a large number of packets to converge.

ICMP Traceback Messages [7]: This method is similar to the probabilistic packet-marking technique, but instead of marking the packets, routers send newly proposed ICMP messages to the destination, with the information about the previous hop. The scheme proposes sending an ICMP message for every 20,000 packets forwarded. Thus the overhead for the scheme is minimal, but it also gives complete path information only after forwarding multiple packets. In this scheme, the attacker can also generate bogus ICMP traceback messages to confuse the victim. Therefore, there must be some provision for verifying the authenticity of an ICMP message without incurring too much overhead.

3.2.3 Rate-Limiting Mechanisms

Rate-limiting mechanisms limit the rate of packet arrivals that match the criteria for DDoS attacks. It is important that rate-limiting mechanisms only limit the rate of malicious packets and do not harm legitimate flows. Furthermore, these rate-limiting mechanisms should not incur a lot of extra overhead, nor should they become a source of denial of service attacks themselves. Rate-limiting attacks can also be thought of as a less severe form of packet filtering. If it is known that the attack-detection mechanism can come up with many false positives, it is better to go for rate limiting rather than packet filtering. A novel aspect is to rate limit aggregates rather than IP source addresses. Aggregates are defined as a subset of traffic defined by some characteristic like a particular destination address. They can be classified on the basis of diverse criteria such as UDP/TCP source ports or IP destination addresses. Routers detect aggregates overwhelming it by using samples of packet drops in the queues. They then send a pushback message to the upstream router along with the information about the aggregate to rate limit and the value of the rate limit. If the aggregate packet respects the rate limit, it is allowed to sail through, otherwise the packets are dropped to conform to the rate limit and pushback messages are recursively propagated to upstream routers.

There are also mechanisms designed for the protection of servers from high traffic rates. One such approach involves a server under stress installing rate throttles at a subset of its upstream routers. On installing such throttles, all the traffic passing through the router to the source S is rate limited to the throttle rate. This scheme shows how to distribute the total capacity of the server in a maximum–minimum fair way among the routers servicing it. This means that only aggressive flows which do not respect their rate shares are punished and not the other flows. There are other rate-limiting schemes that detect bandwidth attacks by noticing an asymmetry between the packets traveling to and from a network. If a host is not replying with as many packets as are being sent to it, this could be an indication of the host being attacked by DoS traffic. The DWARD system is meant to be installed at the edge routers for a network. The system monitors the traffic being sent to and from the hosts in its interior. If it notices asymmetry in the packet rates generated by an internal host, it rate limits the packet rate. Thus, this solution detects at the source (much like egress packet filtering). The downside is the possibility of numerous false positives while detecting DDoS conditions near the source. This is because there might be asymmetry in the packet rates for a short duration. Furthermore, some legitimate flows, such as real-time UDP flows, do exhibit asymmetry.

Multi-level tree for online packet statistics (MULTOPS) is a multilevel data structure that can be used to keep track of asymmetric flows passing through a router. It stores packet-rate statistics for flows between hosts (or subnets) A and B using either A's or B's IP address. When it stores the statistics based on source addresses, it operates in attack-oriented mode, otherwise in the victim-oriented mode. A MULTOPS data structure can thus be used for keeping track of attacking hosts or hosts under attack. When the packet rate to or from a subnet reaches a certain threshold, a new sub-node is created to keep track of packet rates. This process can go on until finally IP address packet rates are being maintained. Therefore, starting from a coarse granularity one can detect with increasingly finer accuracy the exact attack source or destination addresses. The IP source addresses that are obtained are spoofed addresses, but they can still be valuable in applying rate limits.

The hop-count filtering technique is one of the victim-based solutions that can help protect against DDoS attacks. In order to filter packets, the filtering scheme uses the

information contained in the IP header. Although an attacker can forge any field in the IP header, he or she cannot falsify the number of hops an IP packet takes to reach its destination, which is only determined by the Internet routing infrastructure. Since each intermediate router decrements the time to live (TTL) value by one before forwarding a packet, the hop-count information is indirectly reflected in the TTL field of the IP header. The difference between the initial TTL value (at the source) and the final TTL value (at the destination) is the hop-count between the source and the destination. Assuming that attackers cannot disrupt routers to alter TTL values of IP packets, the destination can infer its initial TTL value and therefore the hop-count from the source by examining the TTL field of each arriving packet. This scheme will be explained thoroughly in chapter 4.

Chapter 4

Hop-Count Filtering

4.1 Introduction

The basis for hop-count filtering is that most spoofed IP packets, when arriving at victims, do not carry hop-count values that are consistent with the IP addresses being spoofed. Hop-count filtering builds an accurate IP-to-hop-count mapping table, using pollution-proof methods in order to prevent pollution of the ip to hop-count (IP2HC) mapping table by HCF-aware attackers.

Two running states (alert and action) are used within HCF in order to inspect the IP header of each IP packet. HCF stays in alert state under normal condition, waiting for abnormal TTL behaviors without discarding any packet. Even if a legitimate packet is incorrectly identified as spoofed, it will not be dropped. Hence, there is no collateral damage in alert state. Upon detection of an attack, HCF switches to action state, where it discards packets with mismatching hop-counts. Besides the IP2HC inspection, several efficient mechanisms are available to detect DDoS attacks. Through analysis using network measurement data, HCF can recognize close to 90% of spoofed IP packets [13]. Therefore, HCF discards spoofed IP packets with little collateral damage in action state and thus reduces the percentage of false positives.

4.2 Hop-Count Inspection

The first task of HCF is the validation of the source IP address of each packet via hop-count inspection. In this section, we first discuss the hop-count computation and then detail the inspection algorithm.

4.2.1 Hop-Count Computation

Based on the TTL field, the hop-count information should be computed since it is not directly stored in the IP header. TTL is an 8-bit field in the IP header, originally in-

troduced to specify the maximum lifetime of each packet in the Internet. Before forwarding a packet to the next hop, each intermediate router decrements the TTL value of an in-transit IP packet by one. When a packet reaches its destination, the final TTL value is therefore the initial TTL minus the number of hop counts. The challenge in hop-count computation is that a destination only sees the final TTL value. It would have been simple if all operating systems used the same initial TTL value, but in reality each operating system uses different initial TTL values. Furthermore, we cannot assume a single static initial TTL value for each IP address, since the operating system for a given IP address may change over time.

Fortunately, most modern operating systems use only a few selected initial TTL values: 30, 32, 60, 64, 128, and 255 [4]. This set of initial TTL values covers most of the popular operating systems, such as Microsoft Windows, Linux, and many commercial UNIX systems. We observe that most of these initial TTL values are far apart, except between 30 and 32, 60 and 64, and between 32 and 60. Since Internet traces have shown that few Internet hosts are apart by more than 30 hops, one can determine the initial TTL value of a packet by selecting the smallest initial value in the set that is larger than its final TTL value. For example, if the final TTL value is 100, the initial TTL value should be 128 because it is the smaller of the two possible initial values, 128 and 255. In order to resolve ambiguities in the cases of {30, 32}, {60, 64}, and {32, 60}, the hop-count value for each of the possible initial TTL values should be computed. If there is a match with one of the possible hop counts, the packet is accepted.

The drawback of limiting the possible initial TTL values is that packets from end-systems that use "odd" initial TTL values may be incorrectly identified as having spoofed-source IP addresses. This may happen if a user switches from an operating system that uses a "normal" initial TTL value to one that uses an odd value. Since our filter starts to discard packets only upon detection of a DDoS attack, such end-systems would suffer only during an actual DDoS attack. A study [4] conducted on different operating systems shows that those that use odd initial TTLs are typically older operating systems. Therefore, the benefit of deploying HCF should outweigh the risk of denying service to those end-hosts during attacks because such operating systems constitute a very small percentage of end-hosts in the Internet today.

4.2.2 Inspection Algorithm

The inspection algorithm, which is shown in Figure 8 below, extracts the source IP address and the final TTL value from each IP packet. The algorithm infers the initial TTL value and subtracts the final TTL value from it to obtain the hop count. The source IP address serves as the index to the table to retrieve the correct hop count for this IP address, assuming that an accurate IP2HC mapping table is present. If the computed hop count matches the stored hop count, the packet is legitimate; otherwise, the packet is probably spoofed. However, sometimes a spoofed IP address may happen to have the same hop count as one from a flooding source to the victim. In this case, HCF will not be able to identify the spoofed packet.

It is important to minimize collateral damage under HCF because legitimate packets may be identified as spoofed due to inaccurate IP2HC mapping or a delay in hop-count update. Therefore, an identified spoofed IP packet is only dropped in the action state, whereas HCF only keeps track of the number of mismatched IP packets without discarding any packets in the alert state. This guarantees no collateral damage in the alert state, which should be much more common than in the action state.

```
for each packet:
    extract the final TTL T_f and the IP address S;
    infer the initial TTL T_i;
    compute the hop-count H_c = T_i - T_f;
    index S to get the stored hop-count H_s;
    if  (H_c ≠ H_s)
            the packet is spoofed;
    else
            the packet is legitimate;
```

Figure 8: Hop-count inspection algorithm.

4.3 Running States of HCF

Since HCF causes a delay in the critical path of packet processing, it should not be active at all times. We therefore introduce two running states inside HCF: the alert state to detect the presence of spoofed packets, and the action state to discard spoofed

packets. By default, HCF stays in the alert state and monitors the trend of hop-count changes without discarding packets. Upon detection of a flux of spoofed packets, HCF switches to the action state to examine each packet and discards spoofed IP packets. Having two states can better protect servers against different forms of DDoS attacks.

4.3.1 Tasks in Two States

```
for each sampled packet p:
   spoof = IP2HC_Inspect(p);
   t = Average(spoof);
   if ( spoof )
            if ( t > T1 )
                     Switch_Action();
   Accept(p);

for the k-th TCP control block tcb:
   Update_Table(tcb);
```

Figure 9: Operations in the alert state.

Figure 9 lists the tasks performed in the alert state. In this state, HCF performs the following tasks: sample incoming packets for hop-count inspection, calculate the spoofed packet counter, and update the IP2HC mapping table in case of legitimate hop-count changes. Packets are sampled at exponentially distributed intervals with mean m in either time or the number of packets. The exponential distribution can be precomputed and made into a lookup table for fast on-line access. For each sampled packet, the IP2HC_Inspect () function returns a binary number spoof, depending on whether the packet is identified as spoofed or not. This is then used by the average () function to compute an average spoof counter t per unit time. When t is greater than a threshold $T1$, HCF enters the action state. While in alert state, HCF will also update the HCF table using the TCP control block of every k-th established TCP connection.

```
for each packet p:
   spoof = IP2HC_Inspect(p);
   t = Average(spoof);
   if ( spoof )
         Drop(p);
   else Accept(p);

   if ( t ≤ T2 )
         Switch_Alert();
```

Figure 10: Operations in the action state.

HCF in the action state performs a hop-count inspection for each packet and discards any identified spoofed packets. Operations in the action state are shown in Figure 10. HCF in this state performs similar tasks as in the alert state. The main differences are that HCF must examine every packet and discard spoofed packets. HCF stays in the action state as long as spoofed IP packets are detected. HCF switches back to the alert state when the spoof counter t is greater than a threshold *T2*. For better stability in the simulation, *T2* should be smaller than *T1*. HCF should not alternate between the alert and action states when t varies around *T1*. To minimize the overhead of hop-count inspection and dynamic update in the alert state, their execution frequencies are adaptively chosen to be inversely proportional to the server's workload. A server's workload is measured by the number of established TCP connections. The two thresholds, *T1* and *T2,* should be adjusted on the basis of the server's workload. The general guideline for setting the thresholds according to the server's workload is as follows [13]:

$$Load \nearrow \Rightarrow Rates \searrow \Rightarrow Threshold \searrow$$

These parameters should be user-configurable, depending on the requirement in balancing security and performance of individual networks.

4.3.2 Staying "Alert" to DDoS Attacks

The alert state not only lowers the overhead of HCF, but it also stops other forms of DoS attacks. In DDoS attacks, an attacker fakes IP packets that contain legitimate requests, such as DNS queries, by setting the source IP addresses of these spoofed packets to the victim's actual IP address. The attacker then sends these spoofed packets to a large number of reflectors. Each reflector only receives a moderate flux of spoofed IP packets, so that it may easily sustain the availability of its normal service without causing any alert. The usual intrusion-detection methods based on the ongoing traffic volume or access patterns may not be sensitive enough to detect the presence of such spoofed traffic. In contrast, HCF specifically looks for IP spoofing, so it will be able to detect attempts to fool servers into acting as reflectors. Although HCF is not perfect and some spoofed packets may still slip through the filter, it can detect and intercept enough of the spoofed packets to stop most DDoS attacks.

4.3.3 Blocking Bandwidth Attacks

To protect server resources such as CPU and memory, HCF can be installed at a server itself or at any network device near the servers, such as the firewall. However, this scheme will not be efficient enough to protect against DDoS attacks that target the bandwidth of a network. Protecting the access link of an entire stub network is more difficult because the filtering has to be applied at the upstream router of the access link, which must involve the stub network's ISP.

The difficulty in protecting against bandwidth flooding is that packet filtering must be separated from detection of spoofed packets, since the filtering has to be done at the ISP's edge router. One or more machines inside the stub network must run HCF and actively watch for traces of IP spoofing by always staying in the alert state. For complete protection, the access router should also run HCF in case attacking traffic terminates at the access router. This can be accomplished by substituting a regular end-host configured as a router. In addition, at least one machine inside the stub network needs to maintain an updated HCF table, since only end-hosts can see established TCP connections. Under an attack, this machine should notify the network administrator who then coordinates with the ISP to install a packet filter based on the HCF table on the ISP's edge router.

The two-running-state design makes it natural to separate these two functions—detection and filtering of spoofed packets. Figure 11 shows a hypothetical stub network that hosts a Web server that runs HCF. The stub network is connected to its upstream ISP via an access router and the ISP's edge router. Under normal network conditions, the Web server monitors its traffic and builds the HCF table. When attack traffic arrives at the stub network, HCF at the Web server will notice this sudden rise of spoofed traffic and inform the network administrator via an authenticated channel. The administrator can have the ISP install a packet filter in the ISP's edge router, based on the HCF table. Note that one cannot directly use the HCF table since the hop counts from client IP addresses to the Web server are different from those to the router. Thus, all hop counts need to be decremented by a proper offset equal to the hop count between the router and the Web server. Once the HCF table is enabled at the ISP's edge router, most spoofed packets will be intercepted, and only a very small percentage of the spoofed packets that slip through HCF will consume bandwidth. In this case, having two separable states is crucial since routers usually cannot observe established TCP connections and use the safe update procedure.

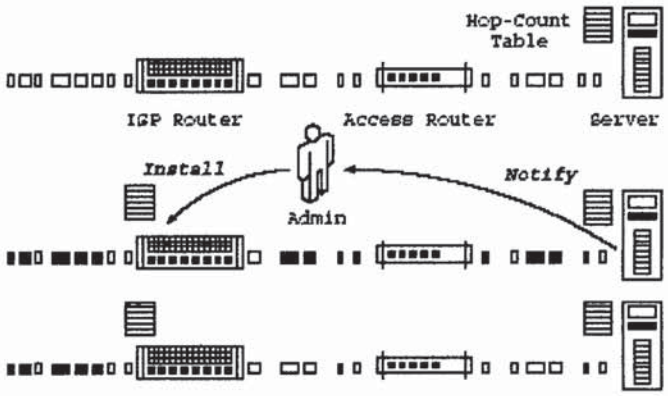

Figure 11: Packet filtering at a router to protect bandwidth.

After describing in detail the hop-count filtering technique, the tools and features used in the simulation will be explained next in chapter 5 along with an analysis of the results.

Chapter 5

Hop-Count Filtering Simulation

5.1 Introduction

In this chapter, we will explain the simulation software known as SSFNet, the overall network system, the computer simulation, and finally the analysis of the simulation results. Using both SSFNet and the hop-count filtering study described in chapter 2, we can reproduce a network simulation showing an actual DDoS attack on servers.

5.2 SSFNet Design Overview

SSFNet is a collection of Java SSF-based components for modeling and simulation of Internet protocols and networks at and above the IP packet level of detail. Link layer and physical layer modeling can be provided in separate components.

SSFNet models are self-configuring, meaning that each SSFNet class instance can autonomously configure itself by querying a configuration database, which may be locally resident or available over the Web. The network configuration files, in the DML format, are used to configure a complete model and instantiate a simulation with the help of the scalable DML configuration database package that is distributed with the SSF simulators.

The principal classes used to construct virtually any Internet model are organized into two derivative frameworks: SSF.OS, used for modeling of the host and operating system components (i.e., protocols) and SSF.Net, used for modeling network connectivity, and creating node and link configurations.

5.2.1 Package SSF.OS

The principal classes of package SSF.OS are:

- ProtocolGraph

- ProtocolSession
- ProtocolMessage
- PacketEvent

This architecture makes it simple to add a new protocol without any changes to the framework, provided its main class is derived from ProtocolSession, and comes with a private DML configuration that extends the base ProtocolSession's DML schema. On shared-memory machines, SSF.OS provides significant performance optimizations by implementing zero-copy message processing and protocol method call chaining.

These classes allow creation of an abstract protocol graph and installation in any combination of protocols. The idea of SSF.OS was originally inspired by the x-kernel, but the design of SSF.OS is simpler, and it differs from the x-kernel in many ways. Internet protocol models are packages built atop SSF.OS, such as SSF.OS.IP, SSF.OS.TCP, SSF.OS.OSPF, and their variants. ProtocolGraph provides methods for establishing proper dependence relations among the installed protocols.

5.2.2 Package SSF.Net

The principal classes of package SSF.Net are:

- Net
- Host and Router
- NIC
- Link

Class Net loads all the model's DML configuration files and controls the orderly instantiation of the entire model: hosts and routers with their protocols, links connecting hosts and routers, as well as traffic scenarios and multiple random number streams. DML configuration is hierarchical and allows recursive composition of larger networks from preconfigured sub-networks.

Class Host is actually derived from SSF.OS.ProtocolGraph, which means that it is fully configurable and can support any network protocol graph. Minimally, this must in-

clude IP, which provides routing and at least one NIC. A Router is a special case of a Host with multiple NICs, and possibly a specialized protocol graph.

Class NIC derived from ProtocolSession is a bottom-level pseudo-protocol that maintains a pair of buffered SSF in/out channels to the world outside the protocol graph. An NIC may be connected to another NIC on the same link, by transparently mapping the encapsulated in/out channels. NIC has various configuration options for physical link characteristics, packet flow buffering, and scheduling, which may be configured from a matching DML description.

Class Link reproduces link-layer connectivity among a set of attached interfaces. A link with more than two attached interfaces implicitly performs level-2 routing of IP packets sent on any attached interface.

5.2.3 DML Network Configuration Database

The DML is a high-level model description, which is simple and standard syntax used for all configuration files. The DML syntax specifies a hierarchy of lists of attributes that can be stored as ASCII files, which are easy to read and interpret.

The DML package, together with the DML standard of model configuration files, support a uniform and efficient approach to the closely related tasks of:

- Model description (configuration)
- Model instantiation in the first phase of a simulation run
- Runtime inspection (reflection)

DML is simple and easy to read and write directly by modelers. At the same time, DML is intended as a target of network design and validation tools, and as a machine-independent model exchange format suitable for storage in databases.

5.3 Overall System Description

In this simulation, we will consider that multiple hosts are trying to connect to multiple servers and getting different responses to their requests. This system is divided into four different networks. Network 0 consists of three routers connecting network 2

and 3 with network 1. Network 1 contains four servers connected to two routers. As for network 2, we have 14 hosts connected to seven routers. And finally, network 3 has 10 hosts connected to four routers. Figure 12 shows how the network system is configured. In total, we have 24 hosts connected to four servers.

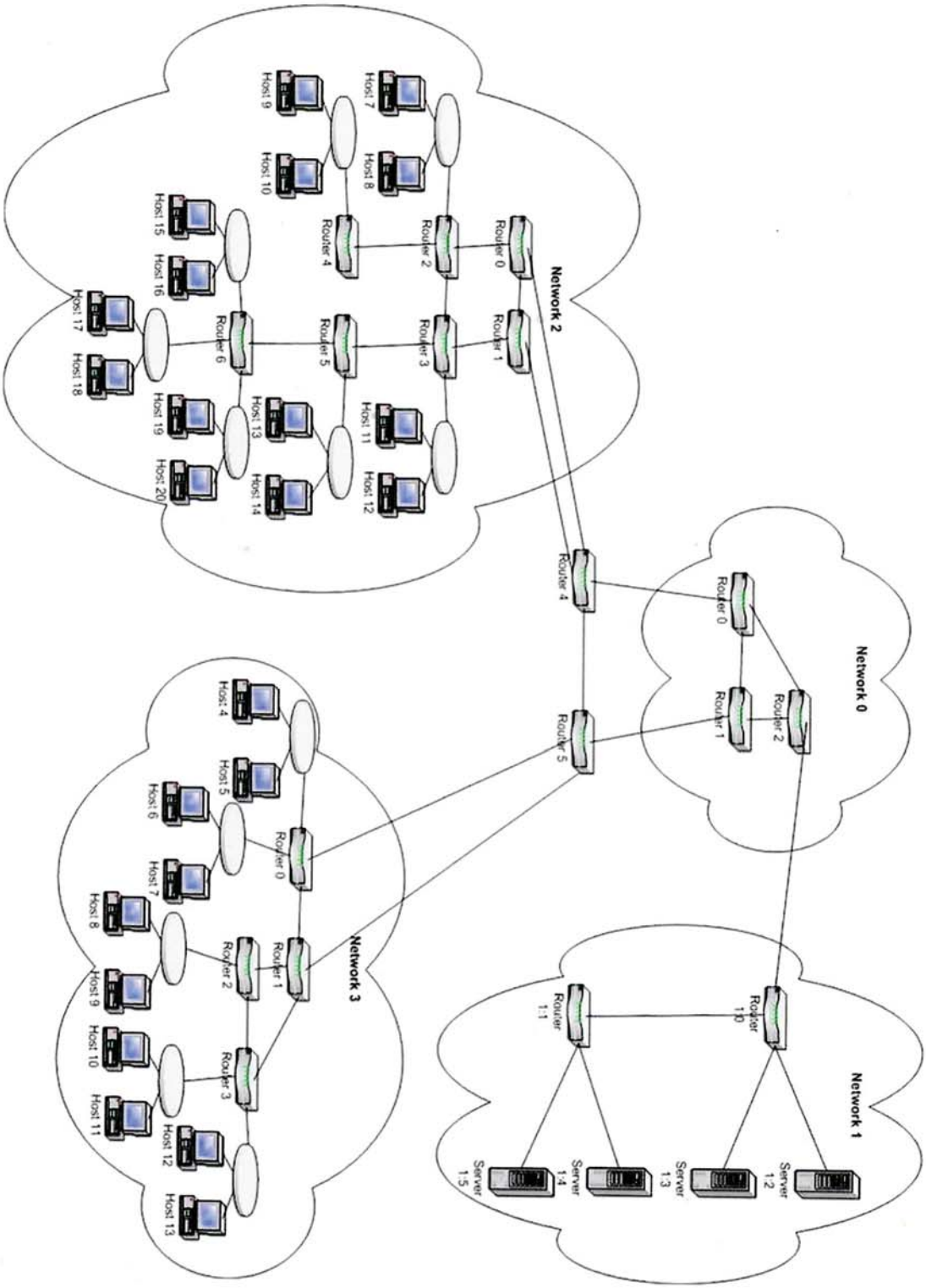

Figure 12: Overall network system.

5.4 Control Logic

The state flow chart that implements the control logic of the HCF states is shown below in Figure 13.

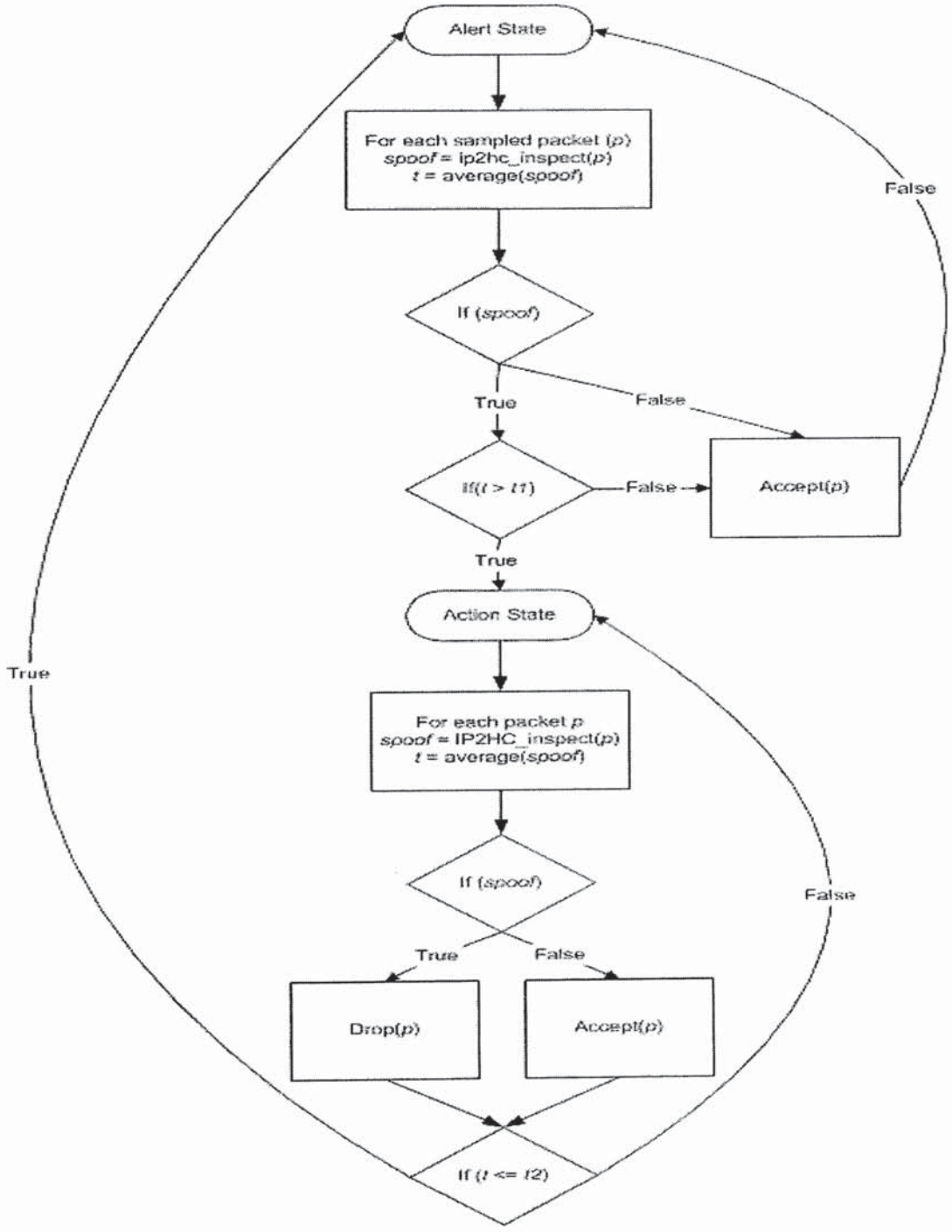

Figure 13: HCF states.

In the alert state, HCF performs the following tasks: sample incoming packets for hop-count inspection, calculate the spoofed packet counter, and update the IP2HC mapping table in case of legitimate hop-count changes. For each sampled packet, the IP2HC_Inspect () function returns whether the packet is identified as spoofed or not. This is then used by the average () function to compute an average spoof counter t per unit time. When t is greater than a threshold $T1$, HCF enters the action state. While in the alert state, HCF will also update the HCF table using the TCP control block of every k-th established TCP connection.

HCF in the action state performs a hop-count inspection for each packet and discards any identified spoofed packets. HCF in the action state performs similar tasks as in the alert state. The main difference is that HCF must examine every packet and discard spoofed packets.

5.5 Simulation Setup

In this section, we will describe all the parameters used in the simulation. We will begin with the thresholds $T1$ and $T2$. Hop-count filtering (HCF) stays in the action state as long as spoofed IP packets are detected. HCF switches back to the alert state when the spoof counter t is greater than a threshold $T2$. $T1$ in this simulation has a value of 5, while $T2$ has a value of 3. $T2$ is smaller than $T1$ for better stability in the simulation because HCF should not alternate between the alert and action states when t varies around $T1$. Furthermore, the server stops at 65 packets per unit time. Whenever the packet load exceeds this threshold, the server is considered to be under a DoS attack, and the hop-count filtering has failed to achieve its objective. Finally, spoofed hosts are chosen randomly from the 24 hosts to test whether the hop-count filtering can help protect servers against DoS attacks. In the simulation, we have increased the number of spoofed hosts from 1 to 12, and we have recorded the results in Table 1.

Spoofed Hosts	Spoofed Packets	Dropped Packets	Total Packets
1	1,004	889	23,095
2	1,007	891	22,094

3	1,010	893	21,093
4	1,013	895	20,092
5	1,016	896	19,091
6	1,019	897	18,090
7	1,022	899	17,089
8	1,025	901	16,088
9	1,028	903	15,087
10	1,031	905	14,086
11	1,031	905	13,085
12	1,034	907	12,084

Table 1: Simulation results.

5.6 Results Analysis

These data shown in Table 2 are calculated from Table 1 where the total number of hosts was 24.

% Host	% Spoofed	% Dropped	% False Negative
4.17	4.35	88.55	0.50
8.33	4.56	88.48	0.53
12.50	4.79	88.42	0.55
16.67	5.04	88.35	0.59
20.83	5.32	88.19	0.63
25.00	5.63	88.03	0.67
29.17	5.98	87.96	0.72
33.33	6.37	87.90	0.77
37.50	6.81	87.84	0.83
41.67	7.32	87.78	0.89
45.83	7.88	87.78	0.96

50.00	8.56	87.72	1.05

Table 2: Simulation analysis.

A false negative occurs whenever a spoofed packet passes through. It is calculated according to this formula:

$$\% \text{ false negative} = \frac{(\text{spoofed packets} - \text{dropped packets}) \times 100}{\text{total packets}}$$

After calculating the false negatives column, the results show that they are negligible. And thus the hop-count filtering technique has minimal collateral damage.

A false positive occurs whenever a legitimate packet is considered spoofed. However, in this simulation, the occurrence of a false positive is nil due to the absence of Network address translation (NAT) network in the simulation where multiple hosts can have the same IP address.

Figure 14 below shows that with an increase of spoofed hosts in this simulation, we have a small decrease in the percentage of packets dropped. The variance between the highest and the lowest percentages, which is 0.83%, is considered to be negligible in comparison with the percentage of spoofed hosts.

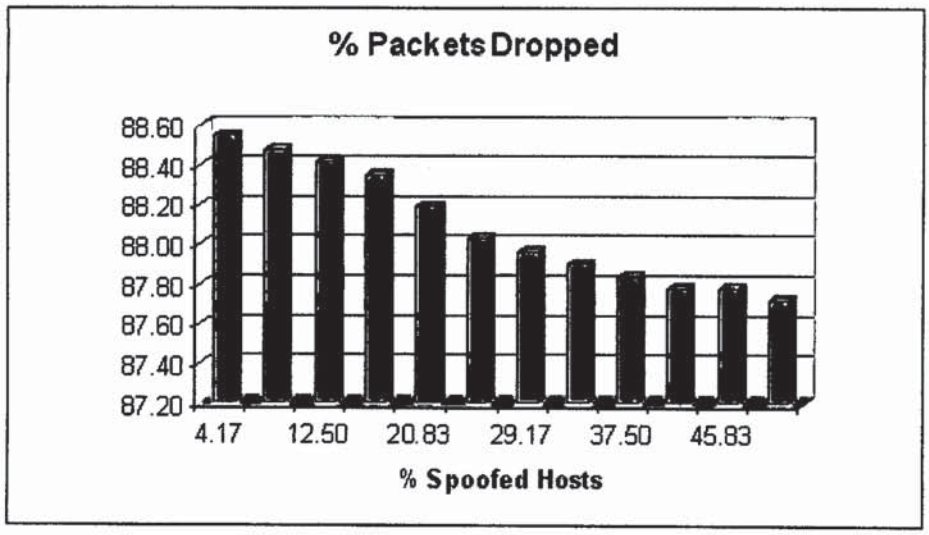

Figure 14: Bar chart showing the percentage of packets dropped.

Hop count is very stable based on the observation that 95% of the paths had fewer than five observable daily changes [13]. Furthermore, the largest percentage of IP addresses that have a common hop-count value is only 10%, which makes it possible for hop-count filtering to work effectively. If the attacker knows the hop count that it takes from the spoofed IP address to the target victim, he/she can fake the initial TTL value accordingly, and hop-count filtering can be evaded. But the DDOS attackers often use a large set of randomly generated IP addresses to carry out attacks, and it is very hard, if not impossible, for the attacker to know all the hop counts between the spoofed IP addresses and the target.

The efficiency of hop-count filtering is based on an IP-to-hop-count table. The hop-count filtering technique shows that the IP-to-hop-count table can be spatially inexpensive when using aggregation techniques such as 24-bit prefix or hop-count clustering. And the table can be safely initialized and kept up-to-date without any pollution.

Hop-count filtering operates in two different states: alert and action. By default, HCF stays in the alert state and monitors the trend of hop-count changes without discarding packets. Upon detection of a spike of spoofed packets, it switches to the action state to examine each packet and discards spoofed IP packets. The advantage of the hop-count filtering idea is simple and has good properties, such as being effective and hard to escape. The two-mode operation (alert and action states) reduces its collateral damage.

However, this proposed approach, which is claimed to be host-based, handles CPU-based DDOS attacks but does not work well for bandwidth-based attacks unless co-operation of the router is provided. It is not clear how long it takes to build from scratch a "fully" functional IP-to-hop-count table. Nor does the technique show what qualifies a good threshold to switch between the alert and action states, or how often the switching-over happens. The 90% effectiveness of hop-count filtering is inferred indirectly from analysis of collected network measurements. However, it would be interesting to see how hop-count filtering behaves in a real-world scenario. Given that the hop-count distribution in the Internet is as it is now: the most common hop-count can be 10%, hop-count filtering is unlikely to be more than 90% effective. Also, it is hard for hop-count filtering to handle cases like NAT, where seemingly the same IP address can be reused by multiple hosts with possibly different hop counts. This is so especially since not every reflector has an incentive to deploy hop-count filtering.

Despite some weaknesses in the hop-count filtering scheme, analysis based on network simulation measurements has shown that 87% of the spoofed traffic can be removed by hop-count filtering. Overall, hop-count filtering is straightforward, easy to implement, readily deployable, and hard to escape.

In conclusion, none of the DDoS detection-and-mitigation schemes can single-handedly protect against the problem, but they can provide useful and powerful tools in alleviating DDoS attacks.

Chapter 6

Conclusion

DDoS attacks are made possible by inherent flaws in the Internet design and the lack of proper security mechanisms in numerous computer systems. Millions of computers are being added to the Internet every year. Many of these systems will be used by home users with permanent IP addresses on a broadband connection. This enriches an already target-rich environment for attackers to scout for systems that can be used as DDoS attack agents. The newer variants of worms like the Code Red worm [2] and the SQL Slammer worm [2] are extremely aggressive and can lead to increased collateral damage. Systems not directly attacked but connected to the attacked systems can also freeze up. This is what happened when many ATM machines stopped working during the time of the Slammer worm attack on January 25, 2003. The situation looks even worse when one considers the possibility that DDoS attacks could become tools of warfare used to target critical infrastructure of other countries. These worms spread at such a fast rate that system administrators cannot be expected to respond in time.

Therefore, there is a need to do more research into tools to automatically scan the network for possible intrusions. The stress heretofore has been to build things that are fast and convenient to use rather than secure. The general perception among the public is also partially responsible for this. The public in many cases is not willing to spend the extra money needed to design secure software. To take an example, an airline whose planes crash once every month is bound to see a huge loss in revenue because of people refusing to use it, but at the same time computer systems that crash much more frequently enjoy a huge market. These perceptions will change over time with security becoming increasingly important to customers as well. It will also not be long before major corporations start holding the software companies responsible for major losses suffered due to security incidents.

References

[1] A Taxonomy of DDoS attacks and DDoS defense Mechanisms. Jelena Mirkovic, Janice Martin and Peter Reiher. Computer Science Department. University of California , Los Angeles, Technical Report #020018.

[2] Distributed Denial of Service (DDoS) Attacks/tools.
Available: http://staff.washington.edu/dittrich/misc/ddos

[3] D. Dittrich. Distributed Denial of Service (DDoS) attacks/tools page.
Available: http://staff.washington.edu/dittrich/misc/ddos/

[4] The Swiss Education and Research Network. Default TTL values in TCP/IP, 2002.
Available: http://secfr.nerim.net/docs/fingerprint/en/ttl default.html

[5] P. Ferguson et. al. RFC 2267. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. Technical report, The Internet Society, 1998.

[6] SANS Institute. Egress filtering v 0.2, 2000.
Available: http://www.sans.org/y2k/egress.htm

[7] Bellovin. ICMP Traceback Messages. Technical report, AT&T, 2000.
Available: http://www.ietf.org/internet-drafts/draft-bellovin-itrace-00.txt

[8] Trends in Denial of Service Technology.
Available: http://www.cert.org/archive/pdf/DoS_trends.pdf

[9] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. ACM SIGCOMM 2001.

[10] K. Park and H. Lee. On the effectiveness of probabilistic packet marking for IP Traceback under a denial of service attack. In Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies, 2001.

[11] David Dittirch. Distributed denial of service (DDoS) Attacks/tools.
Available: http://staff.washington.edu/dittrich/misc/ddos

[12] David Moore, Geoffrey Voelker, and Stefan Savage. Inferring Internet denial of service activity. In Proceedings of the USENIX Security Symposium, Washington, DC, USA, August 2001. USENIX.

[13] Cheng Jin, Haining Wang, Kang G. Shin. Hop-Count Filtering: An Effective Defense Against Spoofed DDoS Traffic. In Proceedings of the 10th ACM conference on Computer and communication security. Washington D.C., USA, 2003.

[14] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the internet. In Proceedings of USENIX Annual Technical Conference '2000, San Diego, CA, June 2000.

Appendix A

```
##############################################################################
# This is the DML configuration file used to describe the network. For each router the
# interfaces are numbered beginning at about "12 o'clock" and increasing clockwise.
##############################################################################

_schema [_find .schemas.Net]

  Net [
    frequency 1000000000
    AS_status boundary
    ospf_area 0

    traffic [
      pattern [
        client 2
        servers [port 10 nhi 1:3(0)]
      ]
      pattern [
        client 3
        servers [port 10 nhi 1:3(0)]
      ]
    ]

    Net [id 0 cidr 0 _extends .networks.network0.Net]
    Net [id 1 cidr 1 _extends .networks.network1.Net]
    Net [id 2 cidr 2 _extends .networks.network2.Net]
    Net [id 3 cidr 3 _extends .networks.network3.Net]

    # router between net 0 and net 2
    router [ id 4
      graphics [x 400 y 300]
      interface [ id 0 _extends .dictionary.1Gb ]
      interface [ id 1 _extends .dictionary.1Gb ]
      interface [ id 2 _extends .dictionary.100Mb ]
      interface [ id 3 _extends .dictionary.100Mb ]
      _find .dictionary.routerGraph.graph
    ]

    # router between net 0 and net 3
    router [ id 5
      graphics [x 700 y 360]
      interface [ id 0 _extends .dictionary.1Gb ]
      interface [ id 1 _extends .dictionary.100Mb ]
      interface [ id 2 _extends .dictionary.100Mb ]
      interface [ id 3 _extends .dictionary.1Gb ]
      _find .dictionary.routerGraph.graph
    ]
```

```
  # this link gets top-level cidr block
  link [cidr 4 attach 4(1) attach 5(3) delay 0.005]

  # links from backbone net 0 to routers 4, 5, 1:0 will get addresses
  # from the block of net 0
  link [cidr 0/3 attach 0:0(2) attach 4(0) delay 0.01]
  link [cidr 0/4 attach 0:1(1) attach 5(0) delay 0.01]
  link [cidr 0/5 attach 0:2(0) attach 1:0(3) delay 0.01]

  # links to connect router 4 to net 2 will get addresses
  # from the block of network2
  link [cidr 2/4 attach 4(2) attach 2:1(0) delay 0.005]
  link [cidr 2/3 attach 4(3) attach 2:0(0) delay 0.005]

  # similarly links to connect router 5 to network3 will get addresses
  # from the cidr block of network3
  link [cidr 3/4 attach 5(2) attach 3:0(0) delay 0.005]
  link [cidr 3/5 attach 5(1) attach 3:1(0) delay 0.005]

  # ospfoptions control the printing of the diagnostic output
  ospfoptions [
   show_ifaces      false # show ospf interface data structure
   show_AS_area     false # show AS and area number
   show_rtr_type    false # show router types
   show_hello_pkts  false # show hello packets and link connection
   show_lsas        false # show lsa packets
   show_lsdb        false # show link state databases
   show_rtg_tbl     false  # show routing tables
  ]
 ]

 networks [

################################################################################
# Defining network 0
################################################################################

  network0 [
   Net [
    router [ id 0
     graphics [ x 300 y 200]
     interface [ id 0 _extends .dictionary.1Gb ]
     interface [ id 1 _extends .dictionary.1Gb ]
     interface [ id 2 _extends .dictionary.1Gb ]    # not attached
     _find .dictionary.routerGraph.graph
    ]
    router [ id 1
     graphics [ x 600 y 200]
     interface [ id 0 _extends .dictionary.1Gb ]
     interface [ id 1 _extends .dictionary.1Gb ]    # not attached
```

```
    interface [ id 2 _extends .dictionary.1Gb ]
    _find .dictionary.routerGraph.graph
   ]
   router [ id 2
    graphics [ x 600 y 100]
    interface [ id 0 _extends .dictionary.1Gb ]    # not attached
    interface [ id 1 _extends .dictionary.1Gb ]
    interface [ id 2 _extends .dictionary.1Gb ]
    _find .dictionary.routerGraph.graph
   ]

   link [ cidr 0 attach 0(1) attach 1(2) delay 0.01] # 10 ms delay
   link [ cidr 1 attach 1(0) attach 2(1) delay 0.01]
   link [ cidr 2 attach 0(0) attach 2(2) delay 0.01]
  ]
 ]

#############################################################################
# Defining network 2
#############################################################################

 network2 [
  Net [
   router [ id 0
    graphics [ x 300 y 400]
    interface [ id 0 _extends .dictionary.100Mb ]   # not attached
    interface [ id 1 _extends .dictionary.100Mb ]
    interface [ id 2 _extends .dictionary.100Mb ]
    _find .dictionary.routerGraph.graph
   ]
   router [ id 1
    graphics [ x 410 y 400]
    interface [ id 0 _extends .dictionary.100Mb ]   # not attached
    interface [ id 1 _extends .dictionary.100Mb ]
    interface [ id 2 _extends .dictionary.100Mb ]
    _find .dictionary.routerGraph.graph
   ]
   router [ id 2
    graphics [ x 300 y 500]
    interface [ id 0 _extends .dictionary.100Mb ]
    interface [ id 1 _extends .dictionary.100Mb ]
    interface [ id 2 _extends .dictionary.100Mb ]
    interface [ id 3 _extends .dictionary.10Mb ]
    _find .dictionary.routerGraph.graph
   ]
   router [ id 3
    graphics [ x 450 y 500]
    interface [ id 0 _extends .dictionary.100Mb ]
    interface [ id 1 _extends .dictionary.100Mb ]
    interface [ id 2 _extends .dictionary.100Mb ]
```

```
 interface [ id 3 _extends .dictionary.100Mb ]
 _find .dictionary.routerGraph.graph
]
router [ id 4
 graphics [ x 300 y 600]
 interface [ id 0 _extends .dictionary.100Mb ]
 interface [ id 1 _extends .dictionary.10Mb ]
 _find .dictionary.routerGraph.graph
]
router [ id 5
 graphics [ x 450 y 600]
 interface [ id 0 _extends .dictionary.100Mb ]
 interface [ id 1 _extends .dictionary.10Mb ]
 interface [ id 2 _extends .dictionary.100Mb ]
 _find .dictionary.routerGraph.graph
]
router [ id 6
 graphics [ x 400 y 700]
 interface [ id 0 _extends .dictionary.100Mb ]
 interface [ id 1 _extends .dictionary.10Mb ]
 interface [ id 2 _extends .dictionary.10Mb ]
 interface [ id 3 _extends .dictionary.10Mb ]
 _find .dictionary.routerGraph.graph
]

# hosts on link 1/5 (lan)
host [ id 7
 graphics [ x 200 y 500]
 nhi_route [dest default interface 0 next_hop 2(3)]
 _extends .dictionary.client10Mb
]
host [ id 8
 graphics [ x 225 y 525]
 nhi_route [dest default interface 0 next_hop 2(3)]
 _extends .dictionary.client10Mb
]

# hosts on link 1/6 (lan)
host [id 9
 graphics [ x 200 y 600]
 nhi_route [dest default interface 0 next_hop 4(1)]
 _extends .dictionary.client10Mb
]
host [id 10
 graphics [ x 235 y 626]
 nhi_route [dest default interface 0 next_hop 4(1)]
 _extends .dictionary.client10Mb
]

# hosts on link 2/2 (lan)
```

```
host [id 11
 graphics [ x 530 y 522]
 nhi_route [dest default interface 0 next_hop 3(1)]
 _extends .dictionary.client10Mb
]
host [id 12
 graphics [ x 560 y 500]
 nhi_route [dest default interface 0 next_hop 3(1)]
 _extends .dictionary.client10Mb
]

# hosts on link 2/4/1 (lan)
host [id 13
 graphics [ x 535 y 630]
 nhi_route [dest default interface 0 next_hop 5(1)]
 _extends .dictionary.client10Mb
]
host [id 14
 graphics [ x 560 y 610]
 nhi_route [dest default interface 0 next_hop 5(1)]
 _extends .dictionary.client10Mb
]

# hosts on link 2/4/3/1 (lan)
host [id 15
 graphics [ x 322 y 718]
 nhi_route [dest default interface 0 next_hop 6(3)]
 _extends .dictionary.client10Mb
]
host [id 16
 graphics [ x 355 y 727]
 nhi_route [dest default interface 0 next_hop 6(3)]
 _extends .dictionary.client10Mb
]

# hosts on link 2/4/3/2 (lan)
host [id 17
 graphics [ x 397 y 736]
 nhi_route [dest default interface 0 next_hop 6(2)]
 _extends .dictionary.client10Mb
]
host [id 18
 graphics [ x 430 y 746]
 nhi_route [dest default interface 0 next_hop 6(2)]
  _extends .dictionary.spoofClient10Mb
]

# hosts on link 2/4/3/3 (lan)
host [id 19    .
 graphics [ x 473 y 769]
```

```
    nhi_route [dest default interface 0 next_hop 6(1)]
    _extends .dictionary.client10Mb
   ]
   host [id 20
    graphics [ x 516 y 758]
    nhi_route [dest default interface 0 next_hop 6(1)]
    _extends .dictionary.client10Mb
   ]

   link [ cidr 1/1 attach 0(1) attach 1(2) delay 0.001 ]
   link [ cidr 1/2 attach 0(2) attach 2(0) delay 0.001 ]
   link [ cidr 1/3 attach 2(1) attach 3(3) delay 0.01 ]
   link [ cidr 1/4 attach 2(2) attach 4(0) delay 0.005 ]
   link [ cidr 1/5 graphics [x 200 y 480] attach 2(3) attach 7(0) attach 8(0) delay 0.003 ]
   link [ cidr 1/6 graphics [x 200 y 580] attach 4(1) attach 9(0) attach 10(0) delay 0.001 ]
   link [ cidr 2/1 attach 1(1) attach 3(0) delay 0.001 ]
   link [ cidr 2/2 graphics [x 530 y 510] attach 3(1) attach 11(0) attach 12(0) delay 0.001 ]
   link [ cidr 2/3 attach 3(2) attach 5(0) delay 0.001 ]
   link [ cidr 2/4/1 graphics [x 535 y 610] attach 5(1) attach 13(0) attach 14(0) delay 0.001 ]
   link [ cidr 2/4/2 attach 5(2) attach 6(0) delay 0.001 ] # added by msc
   link [ cidr 2/4/3/1 graphics [x 322 y 705] attach 6(3) attach 15(0) attach 16(0) delay 0.001 ]
   link [ cidr 2/4/3/2 graphics [x 397 y 716] attach 6(2) attach 17(0) attach 18(0) delay 0.001 ]
   link [ cidr 2/4/3/3 graphics [x 473 y 749] attach 6(1) attach 19(0) attach 20(0) delay 0.001 ]

  ]
  ] # end of network2

################################################################################
# Defining network 3
################################################################################

  network3 [
   Net [
    router [ id 0
     graphics [x 610 y 450]
     interface [ id 0 _extends .dictionary.100Mb ]  # not attached
     interface [ id 1 _extends .dictionary.100Mb ]
     interface [ id 2 _extends .dictionary.10Mb ]
     interface [ id 3 _extends .dictionary.10Mb ]
     _find .dictionary.routerGraph.graph
    ]
    router [ id 1
     graphics [x 800 y 450]
     interface [ id 0 _extends .dictionary.100Mb ]  # not attached
     interface [ id 1 _extends .dictionary.100Mb ]
     interface [ id 2 _extends .dictionary.100Mb ]
     interface [ id 3 _extends .dictionary.100Mb ]
     _find .dictionary.routerGraph.graph
    ]
    router [ id 2
```

```
 graphics [x 712 y 563]
 interface [ id 0 _extends .dictionary.100Mb ]
 interface [ id 1 _extends .dictionary.100Mb ]
 interface [ id 2 _extends .dictionary.10Mb ]
 _find .dictionary.routerGraph.graph
]
router [ id 3
 graphics [x 843 y 563]
 interface [ id 0 _extends .dictionary.100Mb ]
 interface [ id 1 _extends .dictionary.10Mb ]
 interface [ id 2 _extends .dictionary.10Mb ]
 interface [ id 3 _extends .dictionary.100Mb ]
 _find .dictionary.routerGraph.graph
]

# hosts on link 0 (lan)
host [ id 4
 graphics [x 600 y 510]
 nhi_route [dest default interface 0 next_hop 0(3)]
 _extends .dictionary.client10Mb
]
host [ id 5
 graphics [x 620 y 510]
 nhi_route [dest default interface 0 next_hop 0(3)]
 _extends .dictionary.client10Mb
]

# hosts on link 1 (lan)
host [id 6
 graphics [x 650 y 510]
 nhi_route [dest default interface 0 next_hop 0(2)]
 _extends .dictionary.client10Mb
]
host [id 7
 graphics [x 670 y 510]
 nhi_route [dest default interface 0 next_hop 0(2)]
 _extends .dictionary.client10Mb
]

# hosts on link 3/4 (lan)
host [id 8
 graphics [x 710 y 600]
 nhi_route [dest default interface 0 next_hop 2(2)]
 _extends .dictionary.client10Mb
]
host [id 9
 graphics [x 730 y 600]
 nhi_route [dest default interface 0 next_hop 2(2)]
 _extends .dictionary.client10Mb
]
```

```
    # hosts on link 3/5/1 (lan)
    host [id 10
     graphics [x 820 y 650]
     nhi_route [dest default interface 0 next_hop 3(2)]
     _extends .dictionary.client10Mb
    ]
    host [id 11
     graphics [x 840 y 650]
     nhi_route [dest default interface 0 next_hop 3(2)]
     _extends .dictionary.client10Mb
    ]

    # hosts on link 3/5/2 (lan)
    host [id 12
     graphics [x 900 y 700]
     nhi_route [dest default interface 0 next_hop 3(1) ]
     _extends .dictionary.client10Mb
    ]
    host [id 13
     graphics [x 930 y 700]
     nhi_route [dest default interface 0 next_hop 3(1)]
     _extends .dictionary.client10Mb
    ]

    link [ cidr 0 graphics [x 600 y 500] attach 0(3) attach 4(0) attach 5(0) delay 0.001 ]
    link [ cidr 1 graphics [x 650 y 500] attach 0(2) attach 6(0) attach 7(0) delay 0.001 ]
    link [ cidr 2 attach 0(1) attach 1(3) delay 0.01 ]
    link [ cidr 3/1 attach 1(2) attach 2(0) delay 0.003 ]
    link [ cidr 3/2 attach 2(1) attach 3(3) delay 0.003 ]
    link [ cidr 3/3 attach 1(1) attach 3(0) delay 0.003 ]
    link [ cidr 3/4 graphics [x 710 y 590] attach 2(2) attach 8(0) attach 9(0) delay 0.001 ]
    link [ cidr 3/5/1 graphics [x 820 y 630] attach 3(2) attach 10(0) attach 11(0) delay 0.001 ]
    link [ cidr 3/5/2 graphics [x 900 y 690] attach 3(1) attach 12(0) attach 13(0) delay 0.001 ]

   ]
   ] # end of network3

##############################################################################
# Defining network 1
##############################################################################

   network1 [
    Net [
     router [ id 0
      graphics [ x 800 y 100]
      interface [ id 0 _extends .dictionary.1Gb ]
      interface [ id 1 _extends .dictionary.1Gb ]
      interface [ id 2 _extends .dictionary.1Gb ]
      interface [ id 3 _extends .dictionary.1Gb ]   # not attached
```

```
    _find .dictionary.routerGraph.graph
   ]
   router [ id 1
    graphics [ x 800 y 200]
    interface [ id 0 _extends .dictionary.1Gb ]
    interface [ id 1 _extends .dictionary.1Gb ]
    interface [ id 2 _extends .dictionary.1Gb ]
    _find .dictionary.routerGraph.graph
   ]

   host [ id 2
    graphics [ x 900 y 80]
    route [dest default interface 0]
    _extends .dictionary.server1Gb
   ]
   host [ id 3
    graphics [ x 900 y 120]
    route [dest default interface 0]
    _extends .dictionary.server1Gb
   ]
   host [ id 4
    graphics [ x 900 y 160]
    route [dest default interface 0]
    _extends .dictionary.server1Gb
   ]
   host [ id 5
    graphics [ x 900 y 180]
    route [dest default interface 0]
    _extends .dictionary.server1Gb
   ]
   link [ cidr 0 attach 0(2) attach 1(0) delay 0.001 ]
   link [ cidr 1/0 attach 0(0) attach 2(0) delay 0.001 ]
   link [ cidr 1/1 attach 0(1) attach 3(0) delay 0.001 ]
   link [ cidr 2/0 attach 1(1) attach 4(0) delay 0.001 ]
   link [ cidr 2/1 attach 1(2) attach 5(0) delay 0.001 ]
  ]
 ] # end of network1

] # end of networks database

##############################################################################

dictionary [

 # typical nodes
 client10Mb [
  interface [ id 0 _extends .dictionary.10Mb]
  _find .dictionary.clientGraph.graph
 ]
 spoofClient10Mb [
```

```
 interface [ id 0 _extends .dictionary.10Mb]
 _find .dictionary.spoofClientGraph.graph
]
server1Gb [
 interface [ id 0 _extends .dictionary.1Gb]
 _find .dictionary.serverGraph.graph
]

# interface specs
10Mb [
  bitrate 10000000     # 10 million bits per second
  latency 0.001        # 1 ms NIC latency, in seconds (on-card delay)
]
100Mb [
  bitrate 100000000    # 100 million bits per second
  latency 0.0001       # 0.1 ms NIC latency, in seconds (on-card delay)
]
1Gb [
  bitrate 1000000000   # billion bits per second
  latency 0.000001     # 1 us NIC latency, in seconds (on-card delay)
]

# TCP initial parameters
tcpinit [
  ISS 10000           # initial sequence number
  MSS 1000            # maximum segment size
  RcvWndSize  32      # receive buffer size
  SendWndSize 32      # maximum send window size
  SendBufferSize 128    # send buffer size
  MaxRexmitTimes 12    # maximum retransmission times before drop
  TCP_SLOW_INTERVAL 0.5# granularity of TCP slow timer
  TCP_FAST_INTERVAL 0.2# granularity of TCP fast(delay-ack) timer
  MSL 60.0            # maximum segment lifetime
  MaxIdleTime 600.0    # maximum idle time for drop a connection
  delayed_ack false    # delayed ack option
  fast_recovery true   # implement fast recovery algorithm
  show_report false     # print terse TCP connection diagnostics
]

# Three protocol graphs.
routerGraph [graph [
  ProtocolSession [name ip use SSF.OS.IP]
  ProtocolSession [name ospf use SSF.OS.OSPF.sOSPF]
]]

# the client-server protocol needs to agree on some things
appsession [
  request_size  4        # client request datagram size (bytes)
  show_report true        # print client-server session summary report
  debug false           # print verbose client/server diagnostics
```

```
]

clientGraph [graph [
  ProtocolSession [
    name client use SSF.OS.TCP.test.ddos.classes.tcpClient1
    start_time 1.0      # earliest time to send request to server
    start_window 5.0    # send request to server at randomly chosen time
                        # in interval [start_time, start_time+start_window]
    file_size 1000000   # requested file size (payload bytes)
    _find .dictionary.appsession.request_size
    _find .dictionary.appsession.show_report
    _find .dictionary.appsession.debug
  ]
  ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
  ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster
     _find .dictionary.tcpinit]
  ProtocolSession [name ip use SSF.OS.IP]
]]

spoofClientGraph [graph [
  ProtocolSession [
        name spoofClient use SSF.OS.TCP.test.ddos.classes.tcpSpoofClient
    start_time 1.0      # earliest time to send request to server
    start_window 5.0    # send request to server at randomly chosen time
                        # in interval [start_time, start_time+start_window]
    file_size 1000000   # requested file size (payload bytes)
    _find .dictionary.appsession.request_size
    _find .dictionary.appsession.show_report
    _find .dictionary.appsession.debug
  ]
  ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
  ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster
     _find .dictionary.tcpinit]
  ProtocolSession [name ip use SSF.OS.IP]
]]

serverGraph [graph [
  ProtocolSession [
    name server use SSF.OS.TCP.test.ddos.classes.tcpServer1
    port 10          # server's well known port
    _find .dictionary.appsession.request_size
    _find .dictionary.appsession.show_report
    _find .dictionary.appsession.debug
  ]
  ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
  ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster
     _find .dictionary.tcpinit]
  ProtocolSession [name ip use SSF.OS.TCP.test.ddos.classes.serverIP]
 ]]
]
```

Appendix B

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// serverIP is a java file used to simulate the Hop-Count filtering technique.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

package SSF.OS.TCP.test.ddos.classes;

import java.sql.*;

import SSF.OS.*;
import SSF.Net.*;
import SSF.Net.Util.*;
import com.renesys.raceway.DML.*;

public class serverIP extends IP{

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Declaration of variables needed for the simulation
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

    private String url;
    private Connection connect;
    private boolean opened = false;
    private int counter = 0;
    private boolean output = false;

    private boolean first = true;
    private float prevTime = 0;
    private int packetCounter = 0;
    private boolean stopServer = false;
    private float T1 = 0.5f;
    private float T2 = 0.3f;
    private int action_state = 0;

    private String error1 = "";
    private String error2 = "";
    private String error3 = "";
    private String error4 = "";
    private String error5 = "";

    private int c = 0;

    private float countSpoof1 = 0;
    private float countPacket1 = 0;
    private boolean firstSpoof = false;

    private float countPacket2 = 0;
    private float countSpoof2 = 0;
```

```
    /** Host entity where this pseudo-protocol is installed */
    public Host localHost;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// openDB is a function that creates a database in access where the results of the simulation //
can be stored
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public void openDB()
  {
    localHost = (SSF.Net.Host)inGraph();
    // Set up database connection
    try {
      url = "jdbc:odbc:Server Info";

      Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
      connect = DriverManager.getConnection( url );
      System.err.println( "Database Connection successful\n" );
      opened=true;
    }
    catch ( ClassNotFoundException cnfex ) {
      // process ClassNotFoundExceptions here
      cnfex.printStackTrace();
      System.err.println( "Database Connection unsuccessful\n" +
                cnfex.toString() );
    }
    catch ( SQLException sqlex ) {
      // process SQLExceptions here
      sqlex.printStackTrace();
      System.err.println( "Database Connection unsuccessful\n" +
                sqlex.toString() );
    }
    catch ( Exception ex ) {
      // process remaining Exceptions here
      ex.printStackTrace();
      System.err.println( ex.toString() );
    }
  }

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// AddRecord is a function that adds records of data into the database
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

public void AddRecord(int id, float currentTime, String src, String dest, int ttl, ProtocolMessage
payload, String err1, String err2, String err3, String err4, String err5)
  {
     try {
      Statement statement = connect.createStatement();

        String query = "INSERT INTO packet (" +
```

```
            "id, currentTime, src, dest, ttl, payload, error1, error2, error3, error4, error5 " +
            ") VALUES ('" +
            id + "', '" +
            currentTime + "', '" +
            src + "', '" +
            dest + "', '" +
            ttl + "', '" +
            payload + "', '" +
            err1 + "', '" +
            err2 + "', '" +
            err3 + "', '" +
            err4 + "', '" +
            err5 + "')";
      if (output == true)
          System.err.println("\nSending query: " +
                    connect.nativeSQL( query )
                    + "\n" );
          int result = statement.executeUpdate( query );
      if (output == true)
      {
          if ( result == 1 )
             System.err.println("\nInsertion successful\n");
          else
             System.err.println("\nInsertion failed\n");
       }
        statement.close();
      }
      catch ( SQLException sqlex ) {
        sqlex.printStackTrace();
         System.err.println( sqlex.toString() );
      }
  }
```

//

// ip2hc is a function that creates the ip2hc table which is used to check whether a packet is // spoofed or not

//

```
public void ip2hc(int id, String src, int ttl)
  {
    int hopCount1 = 0;
    int hopCount2 = 0;
    if ((ttl > 128) && (ttl <= 255))
{ hopCount1 = 255 - ttl;
 hopCount2 = 0;}
    else if ((ttl > 64) && (ttl <= 128))
{ hopCount1 = 128 - ttl;
 hopCount2 = 0;}
    else if ((ttl > 32) && (ttl <= 64))
{ hopCount1 = 64 - ttl;
```

```
 hopCount2 = 60 - ttl;}
   else if (ttl <= 32)
{ hopCount1 = 32 - ttl;
 hopCount2 = 30 - ttl;}

try {
       Statement statement = connect.createStatement();

       String query = "INSERT INTO ip2hc (" +
           "id, ip, hopCount1, hopCount2 " +
           ") VALUES ('" +
           id + "', '" +
           src + "', '" +
           hopCount1 + "', '" +
           hopCount2 + "')";
     if (output == true)
         System.err.println("\nSending query: " +
                   connect.nativeSQL( query )
                   + "\n" );
        int result = statement.executeUpdate( query );
     if (output == true)
     {
        if ( result == 1 )
           System.err.println("\nInsertion successful\n");
        else
           System.err.println("\nInsertion failed\n");
      }
       statement.close();
     }
     catch ( SQLException sqlex ) {
       sqlex.printStackTrace();
        System.err.println( sqlex.toString() );
     }
  }
```

//

// ip2hc_inspect is a function that inspects a packet according to the ip2hc table whether it
// is spoofed or not

//

```
boolean ip2hc_inspect(String src, int ttl)
{
    int hopCount1 = 0;
    int hopCount2 = 0;
    boolean spoof = false;
    if ((ttl > 128) && (ttl <= 255))
{ hopCount1 = 255 - ttl;
     hopCount2 = 0;}
    else if ((ttl > 64) && (ttl <= 128))
{ hopCount1 = 128 - ttl;
```

```
     hopCount2 = 0;}
    else if ((ttl > 32) && (ttl <= 64))
   { hopCount1 = 64 - ttl;
     hopCount2 = 60 - ttl;}
    else if (ttl <= 32)
{ hopCount1 = 32 - ttl;
     hopCount2 = 30 - ttl;}

//index S to get the stored hop-count Hs
try {
  Statement statement = connect.createStatement();
String query = "SELECT hopCount1, hopCount2 FROM correctip2hc " +
                "WHERE ip = '" +
                 src + "'";
  ResultSet result = statement.executeQuery ( query );
result.next();
  int Hc1 = result.getInt(1);
  int Hc2 = result.getInt(2);

// WARNING Need to take into consideration both hopcounts
//if ((Hc1 == hopCount1) || (Hc1 == hopCount2) || (Hc2 == hopCount1) || (Hc2 == hopCount2))
if (Hc1 == hopCount1)
spoof = false;
else
spoof = true;

  statement.close();
}

     catch ( SQLException sqlex ) {
 ////////        sqlex.printStackTrace();
 ////////        System.err.println( sqlex.toString() );
     }
  return spoof;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// alertState is a function that simulates the alert state of the hop-count filtering scheme
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void alertState(String src, int ttl, boolean spoof, float time)
{
//for each sampled packet
//boolean spoof = ip2hc_inspect(src, ttl);

  if (spoof)
              {
               countSpoof1 = countSpoof1 + 1;
               firstSpoof = true;
              }
```

```
 else
            {
             action_state = 0;
             error2 = "Alert State";
             error3 = "";
             error4 = "";
            }

 if (firstSpoof)    countPacket1 = countPacket1 + 1;

 float t1 = countSpoof1 / countPacket1;

if ((countPacket1 > 10) && (t1 > T1))
            {
             action_state = 1;
             error2 = "";
             error3 = "Action State";
             error4 = "Packet Dropped";
             countSpoof1 = 0;
             countPacket1 = 0;
             firstSpoof = false;
            }
 if (t1 < T1)
            {
             countSpoof1 = 0;
             countPacket1 = 0;
             firstSpoof = false;
            }

//for the k-th TCP control block tcb;
//update_table(tcb);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// actionState is a function that simulates the action state of the hop-count filtering scheme
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void actionState(String src, int ttl, boolean spoof)
{
//for each packet p
//boolean spoof = ip2hc_inspect(src, ttl);

countPacket2 = countPacket2 + 1;

if (spoof)
      {
       error4 = "Packet Dropped";              //drop(p);
       countSpoof2 = countSpoof2 + 1;
       if (packetCounter > 0) packetCounter = packetCounter - 1;
      }
```

```
else
        {
         error4 = "Packet Passed";              //accept(p);
        }

float t2 = countSpoof2 / countPacket2;

if ((countPacket2 > 5) && (t2 <= T2))
            {
             action_state = 0;
             countSpoof2 = 0;
             countPacket2 = 0;
            }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This part is the same as the original serverIP code
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

 private boolean DEBUG  = false;

 // Class implementing IpMonitor used to monitor and/or mark IP packets
 private ProtocolMonitor monitor = null;

 // The ProtocolMonitor on/off switch */
 private boolean monitorON = false;

 public void config(Configuration cfg) throws configException {
  super.config(cfg);
  String str;

  str = (String)cfg.findSingle("debug");
  if (str != null)
    DEBUG = Boolean.valueOf(str).booleanValue();

  Configuration mConfig = (Configuration)cfg.findSingle("monitor");
  if (mConfig != null) createMonitor(mConfig);

  str = (String)cfg.findSingle("tiebreaker.use");
  if (str != null) tieBreakerClass =  str;
 }

 /** Instantiate and configure an IP packet monitor implementing
  *  the interface SSF.OS.ProtocolMonitor, if specified
  *  locally in this host's IP configuration.
  */
 private void createMonitor(Configuration config) throws configException {

  String monitor_type = (String)config.findSingle("use");
```

```
  if (monitor_type == null)
    throw new configException("\n"+debugIdentifier()+" DML monitor.use is not specified");

  try {
    Class mClass = Class.forName(monitor_type);
    Object mobj = mClass.newInstance();
    monitor = (ProtocolMonitor)mobj;
  }
  catch (Exception any) {
    System.err.println(debugIdentifier()+" Can't create an instance of "+monitor_type);
    any.printStackTrace();
    throw new configException("Can't create monitor "+monitor_type);
  }

  //let the monitor do its own config, and let it know the owner IP session
  monitor.config(this, config);

  //turn on the monitor
  setMonitorEnable(true);
 }

 /** Monitor may need to initialize after the config() phase. */
 public void init() throws ProtocolException {
  if (monitor != null) monitor.init();
 }


 /** An ProtocolMonitor may turn on and off calls to its receive()
  *  method. NOTE: the ProtocolMonitor MUST explicitely set
  *  enableMonitor(true) in its init() method to begin
  *  receiving IP packets.
  */
  public void setMonitorEnable(boolean en) {
    if(monitor != null) monitorON = en;
  }

 /** An ProtocolMonitor may inquire if IP calls to its receive() are enabled */
 public boolean getMonitorEnable() {
  return monitorON;
 }

 public boolean push(ProtocolMessage message, ProtocolSession fromSession)
                                        throws ProtocolException {
  IpHeader ipHeader = (IpHeader) message;

   // 1. If the local host is the destination indicated by the ipHeader,
   //    push the payload up to the correct protocol.

   boolean isLocal = // loopback address 127.0.0.1 with netmask 255.0.0.0
(0x7F000000 == (0xFF000000 & ipHeader.DEST_IP));
```

```
   for (int n=0; !isLocal && n<INTERFACE_COUNT; n++)
     isLocal = (INTERFACE_SET[n].ipAddr == ipHeader.DEST_IP);

   if (isLocal) {

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This part is used to stop a server in case of a DDoS attack
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

if (opened == false) openDB();

 // Begin of condition to stop server
 if (stopServer == false)
 {
 counter = counter + 1;
 float currentTime = localHost.now()/(float)SSF.Net.Net.seconds(1.0);
 String src = IP_s.IPtoString(ipHeader.SOURCE_IP);
 String dest = IP_s.IPtoString(ipHeader.DEST_IP);
 int ttl = ipHeader.TIME_TO_LIVE;

 ProtocolMessage payload = message.payload();

 if (first == true)
                {
                 prevTime = currentTime;
                 first = false;
                }
 float diffTime = currentTime - prevTime;

 try { // begin of try
   Statement statement = connect.createStatement();

  String query = "SELECT hopCount1, hopCount2 FROM ip2hc WHERE ip = '" + src +
"'";

    ResultSet result = statement.executeQuery ( query );

    boolean moreRecords = result.next();

    int Hc1 = 0; int Hc2 = 0;

    if (moreRecords)
          { Hc1 = result.getInt(1);
            Hc2 = result.getInt(2);}

    // WARNING :: Must be changed in case the default ttl has changed
    int HcCompare1 = 128 - ttl;
    int HcCompare2 = 0;
```

```
   if ((!moreRecords)||(Hc1 != HcCompare1) || (Hc2 != HcCompare2))
                        { c = c + 1;
                          ip2hc(c, src, ttl);}
   statement.close();

   } //End of try
   catch ( SQLException sqlex ) {
          sqlex.printStackTrace();
          System.err.println( sqlex.toString() );}

   boolean testSpoof = ip2hc_inspect(src, ttl);

   if ((diffTime > 1.0f) && (packetCounter > 65))
                  {
                   error5 ="Server Stopped: DDos attack";
                   stopServer = true;
                  }
     else
                  { error5 = "";
                   error1 = ""; }

   if ((diffTime > 1.0f) && (packetCounter <= 65))
                  { prevTime = currentTime;
                   packetCounter = 0; }

   if (testSpoof == true)  {
                     error1 = "Spoof";
                     packetCounter = packetCounter + 1;
                    }

   if (action_state == 0)  alertState(src, ttl, testSpoof, currentTime);
   else                    actionState(src, ttl,testSpoof);

 AddRecord(counter, currentTime, src, dest, ttl, payload, error1, error2, error3, error4,
error5);

 }  // End of condition to stop server

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This part is the same as the original serverIP code
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

     ProtocolSession handler = session_cache.demux(ipHeader.PROTOCOL_NO);
     if (handler != null) {
      if (monitorON)
        monitor.receive(ipHeader, fromSession, handler);
return (handler.push(ipHeader.payload(), this));
     } else {
if (!ICMP.suppressReporting(ipHeader))
   push(ICMPHeader.makeProtocolUnreachableMessage(ipHeader).
```

```
ipHeader(),this);
      drop(ipHeader);
      return false;
     }
    }

    // 2. If the message has reached the end of time-to-live, drop it.
    // if (0 >= --ipHeader.TIME_TO_LIVE) {  bugfix 12-13-01
    if (0 >= ipHeader.TIME_TO_LIVE--) {
     if (!ICMP.suppressReporting(ipHeader))
  push(ICMPHeader.makeTimeExceededMessage(ipHeader).ipHeader(),this);
     drop(ipHeader);
     return false;
    }

    // 3. If this host is not the destination of this ipHeader, consult
    //    the forwarding table and send it on the proper network interface.
    RoutingInfo rtgInfo =
ROUTING_TABLE.findBest(ipHeader.SOURCE_IP,ipHeader.DEST_IP);

    int recurse = 0;
    while (rtgInfo != null && rtgInfo.next_hop_interface() == null) {
     recurse++;
     if (recurse > 10) { // recursive lookup limit
      throw new Error("IP: recursive lookup limit exceeded");
     }
     rtgInfo=ROUTING_TABLE.findBest(ipHeader.SOURCE_IP,rtgInfo.next_hop_ip());
    }

    if (rtgInfo != null) {
     NIC use_iface = rtgInfo.next_hop_interface();

     // if no source is specified in the header, then set it to the
     // interface which it is about to be sent out on
     if (ipHeader.SOURCE_IP < 0) {
      if (DEBUG) {
       System.err.println(debugIdentifier() + "invalid IP source address: "
                  + ipHeader.SOURCE_IP);
      }
      ipHeader.SOURCE_IP = use_iface.ipAddr;
     }

     ipHeader.NEXT_HOP_IP = rtgInfo.next_hop_ip();

     if (monitorON)
      monitor.receive(ipHeader, fromSession, use_iface);

     return (use_iface.push(ipHeader,this));
    } else {
     if (DEBUG) {
```

```
      System.out.println(debugIdentifier() + "no route for "
                    + IP_s.IPtoString(ipHeader.DEST_IP) + " in:");
      ((RadixTreeRoutingTable)ROUTING_TABLE).print("  ");
    }
    // shouldn't drop() be called here? -bjp 2000.06.29
    if (!ICMP.suppressReporting(ipHeader))
  push(ICMPHeader.makeHostUnreachableMessage(ipHeader).
     ipHeader(),this);
    drop(ipHeader);
    return false;

  }
 } // end of push()
}
```

Appendix C

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// tcpSpoofClientSession is a java file that enables us to change the IP of a packet in order
// to make it spoofed.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

package SSF.OS.TCP.test.ddos.classes;

import SSF.OS.*;
import SSF.OS.Socket.*;
import SSF.OS.TCP.*;
//import SSF.OS.TCP.test.DDos.classes.*;
import SSF.Net.*;
import SSF.Net.Util.*;
import SSF.Util.Streams.*;
import com.renesys.raceway.DML.*;
import java.io.*;
import java.util.*;

/** A simple prototype class executing one TCP client-server session;
 *  it is instantiated by SSF.OS.TCP.test.tcpSpoofClient.
 */
public class tcpSpoofClientSession extends ProtocolSession {
  public static final int CLOSED     = 0x0;
  public static final int CONNECTING = 0x1;
  public static final int REQUESTING = 0x2;
  public static final int READING    = 0x4;
  public static final int CLOSING    = 0x8;

  public static final String stateString(int s) {
  switch(s) {
    case CLOSED: return "CLOSED";
    case CONNECTING: return "CONNECTING";
    case REQUESTING: return "REQUESTING";
    case READING: return "READING";
    case CLOSING: return "CLOSING";
    default: return "?UNKNOWN:"+s;
  }
  }

  /****************************** Class Variables ****************************/

  /** client who created this session */
  public tcpSpoofClient owner;

  /** client-assigned session id */
  public int id;

  /** current session state **/
```

```
public int state = CLOSED;

/** client's Continuation for this session */
public Continuation clientCont;

/** print out debug information */
public boolean showDebug;

/** print out summary session report */
public boolean showReport;

/** size (in bytes) of request datagram sent from client to server */
public int requestSize;

/** requested file size */
public int fileSize;
// long fileSize;  bugfix 11/4/00 ato

/** Host entity where this pseudo-protocol is installed */
public Host localHost;

/** local socket used by this session */
public socketAPI sd;

private String srvNHI;
private int srvIP;
private int srvPort;

private String locNHI;
private int locIP;
private int locPort;
private long dataReceived = 0L;
private String str;
private long startTime;

/****************************** Constructors ******************************/

public tcpSpoofClientSession (tcpSpoofClient client, int sessid,
                tcpSpoofClient.serverData serv, int file_size) {
  owner = client;
  localHost = client.localHost;
  id = sessid;
  locNHI = client.localNHI;
  locIP = client.localIP;

  requestSize = client.request_size;
  fileSize = file_size;
  showReport = client.show_report;
  showDebug = client.debug;
```

```
  srvNHI = serv.nhi;
  srvIP  = serv.ip;
  srvPort = serv.port;

 }

 /****************************** Class Methods ******************************/

 public void begin() {
  startTime = localHost.now();

  state=CLOSED;

  try {
   sd = owner.sockms.socket(this, "tcp");
   clientDebug(0," connect()");
   state=CONNECTING;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The line below changes the IP address into 0.0.0.139
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////


   sd.bind(139,666);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
   sd.connect(srvIP, srvPort, new Continuation() {
    public void success() {  // tcp opened connection to server
     sendRequest();
    }
    public void failure(int errno) {
 state=CLOSED;
 clientDebug(errno, " connect() FAILURE");
    }
   });
  } catch (ProtocolException e) {
   System.err.println("tcpSpoofClient: " + e);
  }
 }

 public void begin(Continuation caller) {
  clientCont = caller;
  begin();
 }

 void sendRequest() {
  state=REQUESTING;
  Object[] obj = new Object[1];
  // obj[0] = new Integer((int)fileSize);  bugfix 11/4 ato
  obj[0] = new Integer(fileSize);
```

```
   sd.write(obj, requestSize, new Continuation() {
public void success() {
 clientDebug(0,(" request " + fileSize + "B to " + srvNHI + " OK"));
 readData();
}
public void failure(int errno) {
 state=CLOSED;
 clientDebug(errno, ("request to " + srvNHI));
}
    });
 }

 void readData() {
  state=READING;
  sd.read(fileSize, new Continuation() {
   public void success() {
    if(showReport || showDebug) {
     double duration = (localHost.now() -
                startTime)/(double)SSF.Net.Net.seconds(1.0);
     str = " rcvd " + fileSize + "B at "
         +  ((long)(8*fileSize/duration))/1000.0
         + "kbps - read() SUCCESS ";
     sessionReport(0,str);
    }
state=CLOSING;
    try {
     sd.close(new Continuation() {
      public void success() {
   clientDebug(0," close() OK");
   state=CLOSED;
       }
      public void failure(int errno) {
   clientDebug(errno, " close() FAILURE");
   state=CLOSED;
       }
     });
    } catch (ProtocolException e) {
      System.err.println(e);
    }
    if(clientCont != null)
     clientCont.success();
   }
   public void failure(int errno) {
clientDebug(errno, " read() FAILURE");
state=CLOSED;
   }
  });
 }

 /** preamble to client-side-only diagnostics */
```

```
  public void clientDebug(int errno, String str){
    if (!showDebug) return;

    StringBuffer lineBuffer = new StringBuffer(200);
    lineBuffer.append(localHost.now()/(double)SSF.Net.Net.seconds(1.0))
      .append(" "+owner.name+" ").append(locNHI).append(" ")
      .append(IP_s.IPtoString(locIP)).append(" sid ").append(id).append(str);

    if(errno>0)
      lineBuffer.append(" "+socketMaster.errorString(errno));
    System.err.println(lineBuffer);
  }
  /** preamble to end2end session diagnostics */
  public void sessionReport(int errno, String str){
    StringBuffer lineBuffer = new StringBuffer(200);
    lineBuffer.append(localHost.now()/(double)SSF.Net.Net.seconds(1.0))
      .append(" [ sid ").append(id).append(" start ")
      .append(startTime/(double)SSF.Net.Net.seconds(1.0))
      .append(" ] "+owner.name+" ").append(locNHI).append(" srv ")
      .append(srvNHI).append(str);

    if(errno>0)
      lineBuffer.append(" "+socketMaster.errorString(errno));
    System.err.println(lineBuffer);
  }

  public boolean push(ProtocolMessage message, ProtocolSession fromSession)
    throws ProtocolException {
    return false;
  }
}
```

Appendix D

The table below shows a small portion of the results taken from the simulation.
The ID column is a unique identifier of a packet.
The Time column shows the time when a packet reached the server.
The source column shows the source IP address of the packet. It could be spoofed.
The destination column shows the destination IP address of the packet.
The TTL column shows the time to live of a packet.
Message 1 shows if the packet is spoofed or not.
Message 2 shows if the hop-count filtering is in the alert state.
Message 3 shows if the hop-count filtering is in the action state.
Message 4 shows if the packet passed or dropped in the action state.
Message 5 shows if the server has stopped due to DDoS attack.

ID	Time	Source	Destination	TTL	Message 1	Message 2	Message 3	Message 4	Message 5
21715	8.488565	0.0.0.130	0.0.1.146	120		Alert State			
21716	8.489397	0.0.0.130	0.0.1.146	120		Alert State			
21717	8.49023	0.0.0.130	0.0.1.146	120		Alert State			
21718	8.491061	0.0.0.130	0.0.1.146	120		Alert State			
21719	8.491893	0.0.0.130	0.0.1.146	120		Alert State			
21720	8.492725	0.0.0.130	0.0.1.146	120		Alert State			
21721	8.493557	0.0.0.130	0.0.1.146	120		Alert State			
21722	8.49439	0.0.0.130	0.0.1.146	120		Alert State			
21723	8.495221	0.0.0.130	0.0.1.146	120		Alert State			
21724	8.496054	0.0.0.130	0.0.1.146	120		Alert State			
21725	8.496885	0.0.0.130	0.0.1.146	120		Alert State			
21726	8.497717	0.0.0.130	0.0.1.146	120		Alert State			
21727	8.498549	0.0.0.130	0.0.1.146	120		Alert State			
21728	8.499381	0.0.0.130	0.0.1.146	120		Alert State			
21729	8.500214	0.0.0.130	0.0.1.146	120		Alert State			
21730	8.501045	0.0.0.130	0.0.1.146	120		Alert State			

21731	8.501877	0.0.0.130	0.0.1.146	120	Alert State
21732	8.502709	0.0.0.130	0.0.1.146	120	Alert State
21733	8.503541	0.0.0.130	0.0.1.146	120	Alert State
21734	8.51015	0.0.1.34	0.0.1.146	121	Alert State
21735	8.5109825	0.0.1.34	0.0.1.146	121	Alert State
21736	8.511823	0.0.1.34	0.0.1.146	121	Alert State
21737	8.512654	0.0.1.34	0.0.1.146	121	Alert State
21738	8.513486	0.0.1.34	0.0.1.146	121	Alert State
21739	8.514318	0.0.1.34	0.0.1.146	121	Alert State
21740	8.51515	0.0.1.34	0.0.1.146	121	Alert State
21741	8.515983	0.0.1.34	0.0.1.146	121	Alert State
21742	8.516814	0.0.1.34	0.0.1.146	121	Alert State
21743	8.517647	0.0.1.34	0.0.1.146	121	Alert State
21744	8.518478	0.0.1.34	0.0.1.146	121	Alert State
21745	8.51931	0.0.1.34	0.0.1.146	121	Alert State
21746	8.520143	0.0.1.34	0.0.1.146	121	Alert State
21747	8.520974	0.0.1.34	0.0.1.146	121	Alert State
21748	8.521807	0.0.1.34	0.0.1.146	121	Alert State
21749	8.522638	0.0.1.34	0.0.1.146	121	Alert State
21750	8.52347	0.0.1.34	0.0.1.146	121	Alert State
21751	8.5243025	0.0.1.34	0.0.1.146	121	Alert State
21752	8.525134	0.0.1.34	0.0.1.146	121	Alert State
21753	8.525967	0.0.1.34	0.0.1.146	121	Alert State
21754	8.526509	0.0.1.75	0.0.1.146	122	Alert State
21755	8.526798	0.0.1.34	0.0.1.146	121	Alert State
21756	8.527342	0.0.1.75	0.0.1.146	122	Alert State
21757	8.527631	0.0.1.34	0.0.1.146	121	Alert State
21758	8.528173	0.0.1.75	0.0.1.146	122	Alert State
21759	8.528462	0.0.1.34	0.0.1.146	121	Alert State
21760	8.529006	0.0.1.75	0.0.1.146	122	Alert State
21761	8.529299	0.0.1.34	0.0.1.146	121	Alert State
21762	8.529838	0.0.1.75	0.0.1.146	122	Alert State

21763	8.530131	0.0.1.34	0.0.1.146	121	Alert State
21764	8.530669	0.0.1.75	0.0.1.146	122	Alert State
21765	8.530963	0.0.1.34	0.0.1.146	121	Alert State
21766	8.531503	0.0.1.75	0.0.1.146	122	Alert State
21767	8.531795	0.0.1.34	0.0.1.146	121	Alert State
21768	8.532334	0.0.1.75	0.0.1.146	122	Alert State
21769	8.532627	0.0.1.34	0.0.1.146	121	Alert State
21770	8.533167	0.0.1.75	0.0.1.146	122	Alert State
21771	8.533459	0.0.1.34	0.0.1.146	121	Alert State
21772	8.534291	0.0.1.34	0.0.1.146	121	Alert State
21773	8.535123	0.0.1.34	0.0.1.146	121	Alert State
21774	8.535954	0.0.1.34	0.0.1.146	121	Alert State
21775	8.536165	0.0.1.2	0.0.1.146	121	Alert State
21776	8.536997	0.0.1.2	0.0.1.146	121	Alert State
21777	8.537828	0.0.1.2	0.0.1.146	121	Alert State
21778	8.538661	0.0.1.2	0.0.1.146	121	Alert State
21779	8.539493	0.0.1.2	0.0.1.146	121	Alert State
21780	8.540325	0.0.1.2	0.0.1.146	121	Alert State
21781	8.541164	0.0.1.2	0.0.1.146	121	Alert State
21782	8.541996	0.0.1.2	0.0.1.146	121	Alert State
21783	8.542828	0.0.1.2	0.0.1.146	121	Alert State
21784	8.54366	0.0.1.2	0.0.1.146	121	Alert State
21785	8.544492	0.0.1.2	0.0.1.146	121	Alert State
21786	8.545324	0.0.1.2	0.0.1.146	121	Alert State
21787	8.546171	0.0.1.2	0.0.1.146	121	Alert State
21788	8.547009	0.0.1.2	0.0.1.146	121	Alert State
21789	8.547843	0.0.1.2	0.0.1.146	121	Alert State
21790	8.548676	0.0.1.2	0.0.1.146	121	Alert State
21791	8.549517	0.0.1.2	0.0.1.146	121	Alert State
21792	8.550348	0.0.1.2	0.0.1.146	121	Alert State
21793	8.55118	0.0.1.2	0.0.1.146	121	Alert State
21794	8.552016	0.0.1.2	0.0.1.146	121	Alert State

21795	8.552848	0.0.1.2	0.0.1.146	121		Alert State		
21796	8.553679	0.0.1.2	0.0.1.146	121		Alert State		
21797	8.554513	0.0.1.2	0.0.1.146	121		Alert State		
21798	8.555345	0.0.1.2	0.0.1.146	121		Alert State		
21799	8.556177	0.0.1.2	0.0.1.146	121		Alert State		
21800	8.557009	0.0.1.2	0.0.1.146	121		Alert State		
21801	8.557841	0.0.1.2	0.0.1.146	121		Alert State		
21802	8.558673	0.0.1.2	0.0.1.146	121		Alert State		
21803	8.558936	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21804	8.5595045	0.0.1.2	0.0.1.146	121		Alert State		
21805	8.559768	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21806	8.560337	0.0.1.2	0.0.1.146	121		Alert State		
21807	8.5606	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21808	8.561169	0.0.1.2	0.0.1.146	121		Alert State		
21809	8.561432	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21810	8.562001	0.0.1.2	0.0.1.146	121		Alert State		
21811	8.562264	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21812	8.563096	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21813	8.563928	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21814	8.56476	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21815	8.565592	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21816	8.566424	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21817	8.567256	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21818	8.567358	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21819	8.568191	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21820	8.569022	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21821	8.569855	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21822	8.570686	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21823	8.571591	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21824	8.572424	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21825	8.573256	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21826	8.574171	0.0.0.130	0.0.1.146	120			Action State	Packet Passed

21827	8.575003	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21828	8.575835	0.0.0.130	0.0.1.146	120		Alert State		
21829	8.576667	0.0.0.130	0.0.1.146	120		Alert State		
21830	8.577499	0.0.0.130	0.0.1.146	120		Alert State		
21831	8.578331	0.0.0.130	0.0.1.146	120		Alert State		
21832	8.579164	0.0.0.130	0.0.1.146	120		Alert State		
21833	8.579995	0.0.0.130	0.0.1.146	120		Alert State		
21834	8.580827	0.0.0.130	0.0.1.146	120		Alert State		
21835	8.581659	0.0.0.130	0.0.1.146	120		Alert State		
21836	8.582491	0.0.0.130	0.0.1.146	120		Alert State		
21837	8.5833235	0.0.0.130	0.0.1.146	120		Alert State		
21838	8.584155	0.0.0.130	0.0.1.146	120		Alert State		
21839	8.584987	0.0.0.130	0.0.1.146	120		Alert State		
21840	8.585819	0.0.0.130	0.0.1.146	120		Alert State		
21841	8.586651	0.0.0.130	0.0.1.146	120		Alert State		
21842	8.587483	0.0.0.130	0.0.1.146	120		Alert State		
21843	8.588315	0.0.0.130	0.0.1.146	120		Alert State		
21844	8.589148	0.0.0.130	0.0.1.146	120		Alert State		
21845	8.589978	0.0.0.130	0.0.1.146	120		Alert State		
21846	8.590811	0.0.0.130	0.0.1.146	120		Alert State		
21847	8.591642	0.0.0.130	0.0.1.146	120		Alert State		
21848	8.592475	0.0.0.130	0.0.1.146	120		Alert State		
21849	8.593307	0.0.0.130	0.0.1.146	120		Alert State		
21850	8.593629	0.0.1.34	0.0.1.146	121		Alert State		
21851	8.594461	0.0.1.34	0.0.1.146	121		Alert State		
21852	8.595302	0.0.1.34	0.0.1.146	121		Alert State		
21853	8.596134	0.0.1.34	0.0.1.146	121		Alert State		
21854	8.596966	0.0.1.34	0.0.1.146	121		Alert State		
21855	8.597798	0.0.1.34	0.0.1.146	121		Alert State		
21856	8.598629	0.0.1.34	0.0.1.146	121		Alert State		
21857	8.599462	0.0.1.34	0.0.1.146	121		Alert State		
21858	8.600293	0.0.1.34	0.0.1.146	121		Alert State		

21859	8.60948	0.0.1.75	0.0.1.146	122		Alert State
21860	8.619644	0.0.1.2	0.0.1.146	121		Alert State
21861	8.620477	0.0.1.2	0.0.1.146	121		Alert State
21862	8.621308	0.0.1.2	0.0.1.146	121		Alert State
21863	8.622141	0.0.1.2	0.0.1.146	121		Alert State
21864	8.622972	0.0.1.2	0.0.1.146	121		Alert State
21865	8.623804	0.0.1.2	0.0.1.146	121		Alert State
21866	8.624643	0.0.1.2	0.0.1.146	121		Alert State
21867	8.625476	0.0.1.2	0.0.1.146	121		Alert State
21868	8.6263075	0.0.1.2	0.0.1.146	121		Alert State
21869	8.62714	0.0.1.2	0.0.1.146	121		Alert State
21870	8.627971	0.0.1.2	0.0.1.146	121		Alert State
21871	8.628803	0.0.1.2	0.0.1.146	121		Alert State
21872	8.62965	0.0.1.2	0.0.1.146	121		Alert State
21873	8.630488	0.0.1.2	0.0.1.146	121		Alert State
21874	8.631323	0.0.1.2	0.0.1.146	121		Alert State
21875	8.632154	0.0.1.2	0.0.1.146	121		Alert State
21876	8.632996	0.0.1.2	0.0.1.146	121		Alert State
21877	8.633827	0.0.1.2	0.0.1.146	121		Alert State
21878	8.63466	0.0.1.2	0.0.1.146	121		Alert State
21879	8.635495	0.0.1.2	0.0.1.146	121		Alert State
21880	8.636328	0.0.1.2	0.0.1.146	121		Alert State
21881	8.637159	0.0.1.2	0.0.1.146	121		Alert State
21882	8.637992	0.0.1.2	0.0.1.146	121		Alert State
21883	8.638824	0.0.1.2	0.0.1.146	121		Alert State
21884	8.639656	0.0.1.2	0.0.1.146	121		Alert State
21885	8.640489	0.0.1.2	0.0.1.146	121		Alert State
21886	8.64132	0.0.1.2	0.0.1.146	121		Alert State
21887	8.642153	0.0.1.2	0.0.1.146	121		Alert State
21888	8.642416	0.0.0.139	0.0.1.146	121	Spoof	Alert State
21889	8.642984	0.0.1.2	0.0.1.146	121		Alert State
21890	8.643247	0.0.0.139	0.0.1.146	121	Spoof	Alert State

21891	8.643816	0.0.1.2	0.0.1.146	121		Alert State		
21892	8.644079	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21893	8.644648	0.0.1.2	0.0.1.146	121		Alert State		
21894	8.644911	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21895	8.64548	0.0.1.2	0.0.1.146	121		Alert State		
21896	8.645743	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21897	8.646575	0.0.0.139	0.0.1.146	121	Spoof	Alert State		
21898	8.647408	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21899	8.648239	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21900	8.649072	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21901	8.649903	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21902	8.650736	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21903	8.651567	0.0.0.139	0.0.1.146	121	Spoof		Action State	Packet Dropped
21904	8.6571245	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21905	8.657956	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21906	8.658789	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21907	8.65962	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21908	8.660452	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21909	8.661358	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21910	8.66219	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21911	8.663022	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21912	8.663937	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21913	8.664768	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21914	8.665601	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21915	8.666432	0.0.0.130	0.0.1.146	120			Action State	Packet Passed
21916	8.667265	0.0.0.130	0.0.1.146	120		Alert State		
21917	8.668097	0.0.0.130	0.0.1.146	120		Alert State		
21918	8.668929	0.0.0.130	0.0.1.146	120		Alert State		
21919	8.669761	0.0.0.130	0.0.1.146	120		Alert State		
21920	8.670593	0.0.0.130	0.0.1.146	120		Alert State		
21921	8.671425	0.0.0.130	0.0.1.146	120		Alert State		