A Linux Framework for Firewall Testing

By

Durga Prabhakar
B.E., Mumbai University, 1997

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Durga Prabhakar, 2003

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.
Firewalls represent the first line of defense against intrusive network attacks. Because firewalls are mission-critical, correctness and performance are important issues. Despite this, little emphasis is placed on systematic firewall testing. Most testing is aimed at specific vulnerabilities which are known to exist. We propose a methodology for firewall regression testing. Our test program generates Ethernet traffic, compares the firewall’s observed behaviour with expected behaviour and reports anomalies. Using this approach, we present correctness test suites for 2 different firewalls: a simple hardware firewall, the *AlphaShield* and a more sophisticated software firewall, *iptables*. We also present performance test results for iptables, focusing on delay as a function of frame length for 10 and 100 Mbps links.
CONTENTS

CONTENTS ....................................................................................... iii

List of Figures .................................................................................... vii

ACKNOWLEDGMENTS ....................................................................... viii

1. Introduction ................................................................................... 1
   1.1 What is a firewall? ................................................................. 1
   1.2 How are firewalls used? ......................................................... 1
   1.3 Packet Filtering ................................................................. 2
   1.4 The Send/Expect Paradigm .................................................. 3
   1.5 Thesis Organization ............................................................ 4

2. Frame I/O ..................................................................................... 5
   2.1 Test Setup ............................................................................. 5
   2.2 Frame Creation and Transmission ....................................... 6
      2.2.1 UDP sockets ............................................................... 6
      2.2.2 Raw Packet Sockets ..................................................... 7
      2.2.3 Character Device ......................................................... 8
   2.3 Maximum frame rate .......................................................... 10
   2.4 Test Results ........................................................................ 11

3. Library Support ........................................................................... 12
   3.1 The need for Library Support ............................................... 12
   3.2 The Socket Library functions .............................................. 13
      3.2.1 SSEtherHdrLen .......................................................... 13
      3.2.2 ss_setEthernetHeader() ............................................. 13
      3.2.3 ss_socket() ............................................................... 14
      3.2.4 ss_setPromiscuousMode() ......................................... 14
      3.2.5 ss_bind() ................................................................. 15
      3.2.6 ss_send() ................................................................. 15
      3.2.7 ss_receive() ............................................................ 16
      3.2.8 ss_close() ............................................................... 16
3.3 The Header Utility library ........................................... 17
  3.3.1 ethIP_arphdr .................................................. 17
  3.3.2 IPHDRLEN ........................................................ 18
  3.3.3 setIPcksum() ................................................... 18
  3.3.4 setIPHeader() .................................................. 18
  3.3.5 setUDPHeader() ................................................ 19
  3.3.6 setTCPHeader() ................................................ 19
  3.3.7 setARPHeader() ................................................ 20
  3.3.8 getIPHeader() .................................................. 20
  3.3.9 getUDPHeader() ................................................ 21
  3.3.10 getTCPHeader() ............................................... 21
  3.3.11 getARPHeader() ............................................... 21

3.4 Expect() .............................................................. 22

4. Case Study: AlphaShield .............................................. 24
  4.1 The AlphaShield .................................................. 24
  4.2 Test Plan ........................................................... 25
  4.3 Test Setup .......................................................... 25
  4.4 Test Cases .......................................................... 26
    4.4.1 ARP tests ...................................................... 26
    4.4.2 IP frame tests ............................................... 27
    4.4.3 Active FTP .................................................... 28
    4.4.4 Session Timeout .............................................. 29
    4.4.5 Session Table Capacity .................................... 30
    4.4.6 Denial of Service Attack Tests ............................ 30
  4.5 Conclusion .......................................................... 31

5. Case Study: iptables .................................................. 32
  5.1 Netfilter ........................................................... 32
  5.2 iptables Terminology ............................................. 33
    5.2.1 Rules, Chains, Targets and Policies ....................... 33
    5.2.2 Tables ........................................................ 34
  5.3 Iptables Testing .................................................... 34
    5.3.1 Test Setup .................................................... 34
    5.3.2 Assumptions .................................................. 37
5.3.3 Basic filter Table Match Operations ........................................... 38
  5.3.3.1 Input Interface ................................................................. 38
  5.3.3.2 Output Interface ............................................................... 38
  5.3.3.3 Protocol ......................................................................... 38
  5.3.3.4 Source IP address .............................................................. 39
  5.3.3.5 Destination IP address ....................................................... 40
  5.3.3.6 Fragment ....................................................................... 41
  5.3.4 udp filter Table Match Operations ......................................... 41
    5.3.4.1 Source Port ................................................................. 41
    5.3.4.2 Destination Port .......................................................... 42
  5.3.5 tcp filter Table Match Operations ......................................... 42
    5.3.5.1 TCP flags ................................................................. 42
    5.3.5.2 SYN ......................................................................... 43
  5.3.6 mac filter Table Match Extension ........................................... 43
  5.3.7 multiport filter Table Match Extension .................................... 43
  5.3.8 limit filter Table Match Extension ......................................... 44
  5.3.9 state filter Table Match Extension ....................................... 45
    5.3.9.1 UDP connection tracking ................................................ 46
    5.3.9.2 TCP Connection tracking ................................................ 48
  5.3.10 NAT Table Target Extensions ............................................... 50
    5.3.10.1 SNAT nat Table Target Extension .................................... 51
    5.3.10.2 MASQUERADE nat Table Target Extension ............................ 52
    5.3.10.3 DNAT nat Table Target Extension .................................... 53

5.4 Performance Testing ................................................................. 54
5.5 Conclusions ............................................................................... 57

6. Related Work .............................................................................. 58

7. Conclusions ................................................................................. 61
  7.1 Contributions ............................................................................. 61
  7.2 Future Work .............................................................................. 62

Bibliography ...................................................................................... 63

A. Library Support Code ................................................................. 66
  A.0.1 socketLibrary.h ................................................................. 66
  A.0.2 socketLibrary.c ................................................................. 67
  A.0.3 headerUtility.h ................................................................. 70
  A.0.4 headerUtility.c ................................................................. 73
<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>B. Alpha Shield Specification</td>
<td>76</td>
</tr>
<tr>
<td>C. Alpha Shield Test Suite code</td>
<td>79</td>
</tr>
<tr>
<td>C.0.5 AlphaShield.c</td>
<td>79</td>
</tr>
<tr>
<td>C.0.6 DOS.c</td>
<td>96</td>
</tr>
<tr>
<td>D. iptables Test Suite code</td>
<td>99</td>
</tr>
<tr>
<td>D.0.7 iptables.c</td>
<td>99</td>
</tr>
<tr>
<td>D.0.8 limitTest.c</td>
<td>137</td>
</tr>
<tr>
<td>D.0.9 performanceTest.c</td>
<td>140</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Firewall Operation</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>The SmartBits 6000B</td>
<td>6</td>
</tr>
<tr>
<td>4.1</td>
<td>AlphaShield Test Setup</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>SessionId variables and values</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Netfilter Packet Traversal</td>
<td>33</td>
</tr>
<tr>
<td>5.2</td>
<td>iptables Test Setup</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>Tuple variables and values</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Input Interface test</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Performance Testing at 10 Mbps</td>
<td>56</td>
</tr>
<tr>
<td>5.6</td>
<td>Performance Testing at 100 Mbps</td>
<td>56</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to thank my supervisor, Dr. Daniel Hoffman, for his intellectual and financial support as well as his guidance in writing this thesis.

I would also like to thank my family, my husband in particular, for their ongoing moral support and encouragement.

Lastly, I gratefully acknowledge the financial assistance I received from the University of Victoria during my graduate studies here.
1. Introduction

1.1 What is a firewall?

A firewall is a device or a group of devices that enforces an access control policy between networks [1]. It acts as a barrier between an internal network and the external world (Internet) as shown in Figure 1.1. A firewall could be a hardware device, an operating system service or an application program.

1.2 How are firewalls used?

Firewalls are important because they keep unwanted packets from entering a private network such as an academic or corporate network. Firewalls can be used in the following ways:

1. Packet filtering firewalls work at the IP layer and usually at the transport layer as well. They protect the internal network by making routing decisions after filtering packets based on information in packet headers. These decisions are made on a per-packet basis.

2. Firewalls are entry points to the network being protected. Hence, they have access to all the packets that are exchanged between the network and the Internet. Firewalls can, therefore, be used to log details of network traffic which

![Figure 1.1: Firewall Operation](image-url)
are periodically examined by the system administrator for patterns indicating imminent attacks.

3. Firewalls can also be used as application level gateways (ALG). Rather than just looking at raw packets, ALGs operate at the application level. For example, an ALG might examine every e-mail message being sent/received and make a decision to transmit or discard it depending on its contents [2].

In this thesis, we will use the term firewall to refer exclusively to packet filtering firewalls.

1.3 Packet Filtering

Most firewalls use packet filtering to decide the fate of each frame that they see. Packet filters are software applications that examine the header fields of frames entering or leaving a network interface. They either allow the frame to pass through or discard it, depending on whether or not the header contents meet certain criteria specified in packet filtering rules. Packet filtering rules can be applied at different points in the routing process, e.g., incoming frames are examined just after arrival and outgoing frames are examined just before being transmitted. Most packet filters can be configured. The firewall administrator can run scripts to adjust the filtering rule sets depending on the security policies of the network being protected. For example, a packet filter may be configured to reject incoming frames with a certain source IP address or incoming frames that are fragmented. Some packet filters are more rigid in that their filtering rule sets cannot be changed. Packet filters are also capable of stateful inspection or dynamic packet filtering. Such packet filters base filtering decisions not only on header information of the packet they are currently inspecting, but also on the header contents of previous frames. They typically store information on all open connections in a state table. The amount and format of the information stored varies across packet filters. It is evident that a packet filter that uses stateful inspection is likely to be more powerful than one that works with only static rules.
1.4 The Send/Expect Paradigm

Despite their critical role, firewall systems are usually tested without well-defined and effective methodologies [3]. Most testing is ad-hoc and concentrates on ensuring that specific vulnerabilities are removed. Also, there are no industry standards for firewall testing. It would, therefore, be useful to create a framework that could be used to build test suites for different types of firewalls.

The send/expect paradigm is the foundation of our Linux framework for automated firewall correctness testing. The idea is for a driver (D) to send a frame (F) to the firewall under test and then wait expectantly to see if F satisfies the firewall rules and returns. When F reaches the firewall, there are 2 possibilities:

1. F passes the packet filtering rules of the firewall and gets routed back to D. When D sees F come back, it concludes that F was able to penetrate the firewall.

2. F is discarded by the firewall and D does not see it come back. In this case, D times out and concludes that F is illegal as far as the current rule set of the firewall is concerned.

The use of the Send/Expect paradigm reduces each correctness test to 4 steps:

1. Configure: set firewall rules (for firewalls that support configuration).

2. Create: the frame to be sent.


4. Expect: check to see if the frame returns.

This makes the source code simple to read and understand. Also, the send/expect paradigm can be used to test many different types of firewalls, whether they sit separately on the network or run on the machine they are trying to protect. It also provides us with an automated comparison between expected and actual results immediately after each test case has been run.
1.5 Thesis Organization

Chapter 2 describes our experiments with frame creation and transmission using Linux UDP sockets, raw packet sockets and character devices. Chapter 3 presents our library support for frame creation and transmission/reception using raw packet sockets. The C source code for the libraries has been included in Appendix A. Chapters 4 and 5 describe the application of the send/expect paradigm using the library support to building automated test suites for specific firewalls. Chapter 4 deals with a simple hardware firewall called the AlphaShield. Chapter 5 talks about a more sophisticated software firewall, iptables. Appendix B contains a specification for the AlphaShield. Appendices C and D contain source code for the AlphaShield and iptables test suites respectively.
2. Frame I/O

This chapter explores different methods of creation and the transmission of Ethernet frames from a Linux PC. The goal is to develop a method that allows us flexibility in setting fields such as the Ethernet and IP header fields. This is important because we want to be able to transmit frames with arbitrary Ethernet and IP addresses in order to test firewall rules effectively. Also, other IP header fields such as those concerned with fragmentation need to be set to test the firewall's response to fragmented IP frames. We would also like to see if these methods of frame transmission are able to transmit frames at wire rate by calculating the frame transmission rates over a range of frame sizes for each method. This is important when we want to test performance in the presence of a firewall.

Frames are transmitted using (1) the Linux UDP and Raw Packet socket API and (2) a Linux character device and its device driver.

All frames use Ethernet as their MAC layer protocol and IP as their network layer protocol. Frames are captured using a network performance analysis hardware device called the SmartBits [4]. The SmartBits has multiple network cards. When one of these is configured to run in capture mode, the SmartBits listens to network traffic on the network card, examines each frame it sees and sets a frame arrival timestamp. When the user decides to end the capture mode, the SmartBits displays timestamp information and the frame contents of all frames it captured in a spreadsheet-like format. The SmartBits is extremely accurate in its timestamp measurements (to less than 1 \( \mu \text{sec} \)) and is thus the automatic choice as a frame capture device.

2.1 Test Setup

The experimental setup consists of a Linux PC connected to a network card on the SmartBits. Application programs, written in C, create frames of different sizes and transmit them using a method described above. The frames are transmitted in
a tight loop and are therefore sent at the highest possible rate. With the C program running on the Linux box and the SmartBits running in capture mode, we are able to retrieve frame arrival timestamp information. This can be used to determine frame transmission rates. Also, the C program for a method of transmission requires its own frame creation mechanism. This gives us insight into the degree of control we could have over frame bits. For example, the program that uses UDP sockets for transmission can specify only some of the frame’s IP header fields such as source and destination IP addresses. The remaining IP header fields and all Ethernet header fields cannot be assigned by the program. On the other hand, the program that sends frame using raw sockets has full control over all frame headers.

2.2 Frame Creation and Transmission

As listed above, we use 3 different methods of frame transmission. We will now describe each of these in more detail.

2.2.1 UDP sockets

UDP sockets, also called datagram sockets, operate at the Transport Layer. These sockets can be used to transmit frames that have UDP as their transport layer protocol. These are connectionless sockets, i.e., they do not require connection
establishment before sending a frame. The application program that uses this method for frame transmission follows the sequence below:

1. Create a UDP socket.
2. Bind the UDP socket to a local IP address and UDP port.
3. Create a buffer of data to be transmitted (the size of the buffer depends on the desired frame length).
4. Send 100 frames having the buffer created above as their UDP payload to the listening SmartBits network card.
5. Repeat Steps 3 and 4 for a set of 7 framelengths of 64, 128, 256, 512, 1024, 1280 and 1518 bytes.

Based on the discussion above it is fairly clear that we are not given any control over frame headers. These are added to the UDP payload as the frame travels down the protocol stack. The only parameters we can control are the source and destination IP addresses and port numbers. Even here, there is not much flexibility as the source IP address on the frame has to be the IP address of one of the Ethernet interfaces on the Linux PC.

2.2.2 Raw Packet Sockets

Post 2.0 releases of the Linux kernel support a new protocol family called PF_PACKET [5]. Sockets whose protocol family is PF_PACKET are called packet sockets. This family allows the application to send frames directly to the network card driver, thus avoiding all the usual protocol stack handling. Thus, any packet sent using a packet socket is passed directly to the Ethernet interface. Packet sockets are of 2 types, a datagram packet socket (SOCK_DGRAM) and a raw packet socket (SOCK_RAW). The former leaves it to the kernel to add the Ethernet header to the packet and therefore does not give the application control over fields in the Ethernet header. The datagram packet socket therefore operates at the Data Link Layer. The latter,
however, requires that the application add the Ethernet header to the frame, thus
giving the application complete control over all fields in the Ethernet header. Since
raw packet sockets suit our requirements of having control over all frame headers
better than datagram packet sockets, we use these in our experiments with packet
sockets. The following sequence of steps was followed to transmit UDP frames using
a raw packet socket:

1. Create a raw packet socket.

2. Bind the socket to an Ethernet interface on the Linux machine.

3. Create an Ethernet header, an IP header a UDP header and add them to a data
   buffer to produce a frame buffer of a desired size.

4. Send the frame buffer through the raw packet socket.

5. Repeat Steps 3 and 4 for a set of 7 frame lengths of 64, 128, 256, 512, 1024, 1280
   and 1518 bytes.

Since frames sent using raw packet sockets bypass the standard UDP/IP pro-
tocol stack and are sent directly to the Ethernet driver, frame transmission rates are
likely to be closer to maximum possible values than with UDP sockets. This, along
with the fact that raw sockets allow control over every bit in frame headers makes
them an attractive option for creation and transmission of frames for firewall testing.

2.2.3 Character Device

A character device in Linux is one that can be accessed sequentially as a stream
of bytes, e.g., the text console [6]. A character device driver is a program that
implements the behaviour of the character device. The Linux operating system treats
all devices as files. A character device has an entry in the /dev directory and is
identified by 2 integers called the major number and the minor number. The major
number identifies the driver associated with the device. The minor number is passed
on to the device driver specified by the major number. It is a way for the driver to
identify the device, since a device driver can control several different devices. Device drivers are created as kernel loadable modules (KLMs). They can be dynamically added to or removed from the Linux kernel without the need to rebuild or reboot the kernel. When a device driver is inserted into the kernel, it registers itself by specifying its major number. This creates a mapping between devices and device drivers having the same major numbers. The device driver implements code for device operations like open(), close(), read() and write(). This code is invoked when an application uses the device for I/O.

Within the Linux kernel, all frames are stored in socket buffers. Frames that are associated with sockets at a higher protocol layer are stored as socket buffers in the kernel before being sent. Also, frames to be transmitted from within the Linux kernel can be created as socket buffers and sent to the outgoing queue of the Ethernet interface. Since character devices operate from within the Linux kernel, no socket operations are required.

In our experiments with frame creation and transmission using character devices, we created a character device, loadgen and its device driver charDriver. We also created a C application program, accessDevice, which would use loadgen for I/O by using the usual file system calls provided by Linux. The following sequence of events causes loadgen to create and send Ethernet frames:

1. accessDevice invokes open() on loadgen. This causes the open() function in charDriver to be executed. loadgen is opened and a file descriptor is returned.

2. accessDevice then calls write() using the file descriptor returned by open() above and writes into loadgen a buffer containing the number of frames to be transmitted (n). This causes the write() function in charDriver to be executed. The following events occur in the write() function of charDriver:

   (a) The buffer sent by accessDevice is used to determine n.

   (b) Memory is allocated to allow the creation of n socket buffer structures (struct sk_buff).
(c) The frame header fields of each socket buffer can then be set as needed. `struct sk_buff` contains sub-structures for the frame headers of all layers.

(d) Outgoing Ethernet interface information is entered into the socket buffer by using the sub-structure `net_device` contained in `struct sk_buff`.

(e) The `dev_queue_xmit` function is called with a pointer to the socket buffer to be transmitted as an argument. This causes the socket buffer to be sent to the outgoing queue of the appropriate Ethernet interface, whose driver is then responsible for putting the socket buffer (as a frame) out on the network.

Thus, the use of character devices for frame creation and transmission does not pose any restrictions on frame header field contents. Also, since the character device sends frames directly to Ethernet interface queues, frame transmission rates are expected to be very close to maximum possible values. However, since character device drivers work from inside the Linux kernel, they have to be used more carefully than the other 2 methods that use sockets and work from outside the Linux kernel. This is because they operate in kernel memory space and any error in these devices could cause the kernel to crash.

### 2.3 Maximum frame rate

The maximum frame rate is calculated by dividing the link speed (10/100 Mbps) by the total number of bits that are transmitted per frame sent. In addition to the frame headers and data, the total number of bits transmitted per frame includes a 4-byte cyclic redundancy check (CRC) and an 8-byte preamble. Also, after the transmission of each frame, the sender has to pause for at least 12 byte times to allow for the signal to be propagated to the receiver. This is called the interframe gap (IFG) and adds another 12 bytes to the total number of bits to be transmitted per frame sent. This means that:

$$\text{Maximum frame rate} = \frac{\text{Link Speed}}{\text{Preamble + Frame Size + IFG}}$$
Therefore, for a 64-byte frame, the maximum frame transmission rate for a 10 Mbps link can be calculated as follows:

\[
\text{Maximum frame rate} = \frac{10 \times 10^6}{64 + 512 + 96} = 14,880 \text{ frames/sec}
\]

Thus at 10 Mbps and 100% utilization, wire rate requires 14,880 64-byte frames to be transmitted per second.

2.4 Test Results

The above discussion about maximum frame rates can be extended to state that theoretically, the difference between the timestamps of 2 consecutive 64-byte frames at 10 Mbps is \(\frac{1}{14,880} \text{ sec} = 67.2 \mu\text{sec}\). As described in previous sections, the SmartBits provides us with a spreadsheet containing arrival timestamps for each frame it saw during capture mode. Using this, we can calculate the average experimental maximum frame transmission rate for a frame length and compare it with theoretical values obtained using the formula above.

We found that all 3 methods do very well, with frame transmission rates almost exactly equal to the theoretical values. For 64-byte frames, UDP sockets have frame rates of about 96% of the expected value. For longer frames, however, the frame rates for UDP sockets are the same as those for raw packet sockets and character devices. This is reasonable because, for short frames, the path through the UDP/IP protocol stack is traversed more times per second than for longer frames. Also, there is practically no difference between frame transmission rates using raw packet sockets and character devices. Both these methods give us full control over frame header fields and can therefore be used to generate legal and illegal frames for firewall testing. Since raw packet sockets can be created and used from outside the kernel, they are easier to work with than character devices. This makes raw sockets our method of choice for frame I/O.
3. Library Support

This chapter presents library support for frame creation, transmission, and reception using raw packet sockets.

3.1 The need for Library Support

The Linux kernel provides support for several different types of sockets at various protocol levels. As a result, the socket-related Linux system calls are generic in nature and require several parameters to work correctly. They also require that the program include a large number of C header files which makes the program lengthy and complicated. Since we use a specific type of socket, i.e., the raw packet socket, and use only Ethernet, it is useful to provide a simplified API on top of the one already provided by Linux. This API consists of functions and structures that are specialized for raw packet sockets for Ethernet frame transmission and reception.

The API is written using C and consists of 2 files. The socket library contains structures and function calls needed to create raw Ethernet frames (frames with no network or transport layer headers) and to send/receive them using raw packet sockets. The header utility library contains structures and function calls needed to create frames with IP as their network layer and TCP or UDP as their transport layer.

Therefore, the socket library provides support to create and send a raw Ethernet frame. This is useful, for example, during test set up when we want to check network connectivity. For this, the socket library can be used to write a simple C program that sends a few raw Ethernet frames over the network and verifies that the frames are being received on the other side. The header utility library is required when we want to create and send/receive the more complex frames used in the actual testing process.
3.2 The Socket Library functions

As described earlier, the socket library contains functions that facilitate creation, transmission, and reception of raw Ethernet frames using raw packet sockets. All functions and structures are prefixed by ss_ to distinguish them from the corresponding Linux system calls. Most of them do not have a return value. If an error occurs, they use the standard perror() system call along with errno to display the error number and text.

3.2.1 SSETHERHDRLEN

This is a constant for the size of the Ethernet header and the CRC and is frequently used for calculation of payload lengths required to be set in network and transport layer header length fields. Its value is 18.

3.2.2 ss_setEthernetHeader()

This function sets the fields of the Ethernet header which consists of 6-byte source and destination MAC addresses and a 2-byte protocol. It takes the following parameters:

1. e: a pointer to a standard Linux Ethernet header structure, struct ethhdr. The function assumes that the calling program has allocated memory for *e.

2. dstMAC: a pointer to a buffer containing the destination MAC address of the frame being created.

3. sourceMAC: a pointer to a buffer containing the source MAC address of the frame being created.

4. p: the protocol of the Ethernet payload, i.e., IP or ARP.

This function copies dstMAC, sourceMAC and p into the corresponding fields of the Ethernet header being pointed to by e and returns. Since e is passed by reference, the calling program can use it to insert the required Ethernet header in the frame being created.
3.2.3  ss_socket()

This function creates a raw packet socket and returns its file descriptor. It is thus a specialization of the standard socket() system call provided by the Linux kernel. The socket() system call requires 3 parameters: domain, type and protocol. In our case, the domain is the low level packet interface (protocol family PF_PACKET). Corresponding to this domain, the type of socket we are interested in is the raw socket (SOCK_RAW). The protocol parameter represents the protocol being shipped within the Ethernet frame. It must match one of the registered protocols listed in /usr/include/linux/if ether.h. This parameter is significant when the program is listening for Ethernet frames on the socket. If, for example, the protocol has been set to ETH_P_IP, only IP packets will be allowed through the socket and ARP will not be detected. During our tests, we need to detect both IP and ARP frames. This can be done by setting the protocol parameter to ETH_P_ALL, which means that every Ethernet frame seen at the socket will be captured. ss_socket, therefore, requires no parameters. It opens a socket, using the Linux socket(), whose domain is PF_PACKET, type is SOCK_RAW and protocol is ETH_P_ALL. It then returns the sockets file descriptor to the calling program.

3.2.4  ss_setPromiscuousMode()

Raw packet sockets allow an application to capture frames as they are received on the network card but still do not allow it to retrieve frames not addressed to the host on which the application is being executed [5]. This is because the network card normally retains only frames whose destination MAC address is either its own hardware address or a multicast or broadcast MAC address. The network card can be put into “promiscuous mode” in which it will accept all frames irrespective of their destination MAC addresses. ss_setPromiscuousMode does just this. It takes 2 parameters:

1. fd: the file descriptor of an open socket.

2. i: a pointer to the name of the Ethernet interface that the socket has been
bound to (an Ethernet interface corresponds to a network card).

`ss PromiscuousMode` makes an `setsockopt()` call to the socket (represented by `fd`) which sets it in promiscuous mode.

### 3.2.5 ss.bind()

This function binds a raw socket to an Ethernet interface. It uses the standard Linux `bind()` system call but provides a simpler interface to the user. `ss_bind()` takes the following parameters:

1. **fd**: the file descriptor of the socket that needs to be bound.
2. **i**: a pointer to the name of the Ethernet interface, e.g., `eth0`, that the socket represented by `fd` has to be bound to.

Internally, `ss_bind` calls `bind()` to which it passes 2 main parameters: `fd` and a device independent physical layer address structure called `struct sockaddr_ll`. One of the fields in `sockaddr_ll` is `sll_ifindex`, the interface index of `i`. `sll_ifindex` is determined by making an `ioctl` call to the socket. The `ioctl` call uses the interface name `i` to return its interface index. All this detail, is however, hidden from the socket library user.

### 3.2.6 ss.send()

This function puts the frame on the network. It is the Linux `send()` without the optional `flags` parameter. It takes 3 parameters:

1. **fd**: the file descriptor of the open socket to write the frames to.
2. **buf**: a pointer to the frame buffer to be sent.
3. **len**: the size of `buf`.

The `sendFrame()` function, which is part of the application code, implements the send part of `send/expect` by simply calling `ss_send()`.
3.2.7 ss_receive()

This function reads frames on an open socket and returns an integer containing the number of bytes read or -1 on error. It forms the basis for Expect(). It has 2 input and 2 output parameters. The input parameters include:

1. fd: the file descriptor of the open socket to read.

2. int t: if \( t > 0 \), it represents the time in seconds for which ss_receive must listen before returning because of a timeout (non-blocking receive). If \( t = -1 \), ss_receive blocks forever.

The output parameters include:

1. e: a pointer to the Ethernet header structure (\texttt{struct ethhdr}) of the frame that was received. ss_receive assumes that the calling program has allocated memory for *e.

2. d: a pointer to the Ethernet payload portion of the frame that was received. ss_receive assumes that the calling program has allocated at least 1500 bytes, the largest allowed Ethernet payload excluding CRC, for *d.

ss_receive() uses the Linux select() system call. This helps make it non-blocking with a timeout and ideal for use in Expect(). Once it has read a frame, ss_receive() extracts the Ethernet header and copies it to the memory that e points to. It then extracts the Ethernet payload and copies it to the memory that d points to.

3.2.8 ss_close()

Used to close an open socket, this function is the same as the Linux close() system call. It has been added here for the sake of completeness. It takes one parameter, fd, the file descriptor of the open socket that has to be closed.
3.3 The Header Utility library

This file contains functions needed to set and extract higher level frame headers such as ARP, IP, TCP and UDP headers. The functions that set the header fields have no return value. One of their parameters is a pointer to a structure representing the header to be initialized. The idea here is to provide the user with a partially initialized structure containing values in the important fields such as source and destination IP addresses or ports. It is then up to the user to modify any other fields necessary for special purposes, e.g., the fragmentation-related fields in the IP header or the flags in the TCP header. The same principle applies to the functions that extract frame headers, i.e., they have no return value as such, but one of their parameters is a pointer to a structure containing the header that they have extracted. While the set functions are used during frame creation (before Send), the get functions are used after frame reception (after Expect).

3.3.1 ethIP_arphdr

This structure lays out the ARP header with hardware address fields corresponding to Ethernet and protocol fields corresponding to IP. It has the following fields:

1. arphdr: the standard Linux structure (struct arphdr). This contains fields to specify the hardware address type (ar_hrd, Ethernet in our case), the protocol type (ar_pro, IP in our case), the hardware address length (ar_hln, 6 in our case), the protocol address length(ar_pln, 4 in our case) and the op code (ARP request or ARP reply).

2. ar sip: the IP address of the machine issuing the ARP.

3. ar tip: the IP address of the machine whose hardware address is solicited.

4. ar sha: The hardware (MAC) address of the machine issuing the ARP.

5. ar tha: The hardware (MAC) address of the target machine. This is significant only in an ARP reply.
3.3.2 IPHDRLEN

This is a constant representing the length of the IP header without options. Its value is 20. It is frequently used while setting payload lengths of transport layer headers like TCP and UDP headers.

3.3.3 setIPChecksum()

This function calculates and sets the IP checksum in an IP header. It takes one parameter, *ip, a pointer to the IP header whose checksum needs to be calculated. It first sets the checksum field of *ip to 0. It then calculates the checksum using the standard IP checksum calculation algorithm. Finally, it sets check in *ip to the calculated checksum and returns.

3.3.4 setIPHeader()

This function initializes an IP header structure. We assume that the options portion of the IP header is not being used. The function takes the following parameters:

1. ip: a pointer to a Linux ip header structure (struct iphdr) which is to be initialized. The function assumes that the calling program has allocated memory for *ip.

2. destIP: a pointer to a buffer containing the destination IP address of the frame.

3. sourceIP: a pointer to a buffer containing the source IP address of the frame. Since we are using raw packet sockets, we are allowed to set the source IP address to any arbitrary value.

4. protocol: the transport layer protocol, e.g., IPPROTO_UDP or IPPROTO_TCP.

5. framelen: the length of the frame in bytes. It is used to set the length of the IP payload.

setIPHeader() copies the above parameters into the corresponding fields of *ip. It sets all other IP header fields to default values, e.g., ttl is set to IPDEFTTL and all
fragmentation related fields are set to 0. Finally, it sets the IP header checksum by calling `ss_IPCksum()` on *ip.

### 3.3.5 `setUDPHeader()`

This function is used to initialize a UDP header. It takes the following parameters:

1. `udp`: a pointer to the Linux udp header structure (struct `udphdr`) that has to be initialized. It is again assumed that the calling program has allocated storage for *udp.

2. `destPort`: the destination port number.

3. `sourcePort`: the source port number.

4. `framelength`: the length of the frame in bytes. It is used to set the length of the UDP payload.

The UDP header checksum is set to 0 since all of our testing at this time is confined to the network layer (IP) and the MAC layer (Ethernet) and we do not require transport layer services at the receiving end.

### 3.3.6 `setTCPHeader()`

This function is used to initialize a TCP header. It takes the following parameters:

1. `tcp`: a pointer to the standard Linux TCP header structure (struct `tcphdr`) to be initialized. It is again assumed that the calling program has allocated storage for *tcp.

2. `destPort`: the destination port number.

3. `sourcePort`: the source port number.

Again, the checksum field in the TCP header is set to 0. Also, other TCP header fields such as the sequence number, window size and acknowledgement number are initialized to 0. It is the users responsibility to set these fields in *tcp as needed.
3.3.7 setARPHeader()

This function initializes an ARP header structure. It assigns values to the **ethIP_arphdr** fields that are common to both ARP requests and replies. It is then up to the user to populate the operation code (**ARPOP_REQUEST** or **ARPOP_REPLY**) and the target machine’s MAC address (used only in an ARP reply frame). The function takes the following parameters:

1. **arp**: a pointer to a **ethIP_arphdr** structure. The function assumes that the calling function has allocated space for *arp.*

2. **senderMAC**: the MAC address of the machine that issues the ARP.

3. **senderIP**: the IP address of the machine that issues the ARP.

4. **targetIP**: the IP address of the target machine.

After the call to **setARPHeader**, the user has a partially initialized **ethIP_arphdr** structure which can then be transported as Ethernet payload. For an ARP request, the destination MAC address in the Ethernet header should be broadcast.

3.3.8 getIPHeader()

This function extracts the IP header and returns a pointer to the first byte of the IP payload in the frame. It takes 2 parameters:

1. **ip**: a pointer to an **iphdr** structure. It is assumed that the calling function has allocated enough space for *ip.*

2. **buf**: initially, this is a pointer to the first byte of the IP header to be extracted. After **getIPHeader()** has finished executing, it contains the address of the first byte of IP payload.

Thus, **getIPHeader()** moves the frame pointer over by 20 bytes (size of the IP header).
3.3.9  getUDPHeader()  
This function extracts the UDP header and returns a pointer to the first byte of the UDP payload in the frame. It takes 2 parameters:

1. udp: a pointer to an udphdr structure. It is assumed that the calling function has allocated enough space for *udp.

2. buf: initially, this is a pointer to the first byte of the UDP header to be extracted. After getUDPHeader() has finished executing, it contains the address of the first byte of UDP payload.

Also, before returning, this function converts the source and destination UDP ports in *udp from network byte order to host byte order.

3.3.10  getTCPHeader()  
This function extracts the TCP header and returns a pointer to the first byte of the TCP payload in the frame. It takes 2 parameters:

1. tcp: a pointer to a tcphdr structure. It is assumed that the calling function has allocated enough space for *tcp.

2. buf: initially, this is a pointer to the first byte of the tcphdr to be extracted. After getTCPHeader has finished executing, it contains the address of the first byte of TCP payload.

Also, before returning, it converts the source and destination TCP ports in *tcp from network byte order to host byte order.

3.3.11  getARPHeader()  
This function extracts the ARP header. It takes 2 parameters:

1. arp: a pointer to an ethIP_arphdr structure. It is assumed that the calling function has allocated enough space for *arp.

2. buf: a pointer to the first byte of the ARP header to be extracted.
3.4 **Expect()**

Although `Expect()` is not part of either the socket library or the header utility library, it is worth looking at in some detail here because it is used as the last step of every test case in the following chapters. `Expect()` is considered part of the application code. Users may modify its parameters, structure and return values to suit their specific needs. Our version of `Expect()` takes the following 3 parameters:

1. `fd`: the open socket on which the program should expect to see a frame.

2. `expectedFrame`: the frame that was sent by the driver during the send phase, either as is, or with some modifications to account for network address translation (NAT).

3. `compare`: a pointer to a comparison function which decides if the firewall passed the test. The specific comparison function used depends on the type of frame expected, i.e., the type of `expectedFrame`. If no frame is expected, `compare` is set to NULL.

In our test suites, we provide comparison functions for raw IP (`IPCompare()`), TCP/IP (`TCPCompare()`), UDP/IP (`UDPCompare()`) and ARP (`ARPCompare()`) frames. Therefore, for example, we invoke `Expect()` with `compare` set to the address of `TCPCompare()` if `expectedFrame` is a TCP/IP frame. Our comparison functions take 2 parameters: a pointer to `expectedFrame`, and a pointer to the frame received on `fd` (`actualFrame`). They compare certain header fields of `actualFrame` with the corresponding `expectedFrame` header fields and return 1 if the frames have the same values for the header fields compared. For example, `TCPCompare()` checks if the IP protocol, the source and destination IP addresses and TCP ports of `expectedFrame` and `actualFrame` match. If so, `TCPCompare()` decides that the frame expected was received and returns 1. Users can create their own comparison functions, which are part of the application code.

`Expect()` invokes `ss_receive()`. If `compare` is NULL, i.e., the driver wants to verify that the firewall has dropped the frame, `Expect()` waits for `ss_receive()` to
time out. If, however, `ss_receive()` sees a frame on `fd` before it times out, then `Expect()` returns 0 for error.

If `compare` is not NULL, then `Expect()` waits for `ss_receive()` to return `actualFrame`. If `actualFrame` arrives on `fd` before `ss_receive()` times out, then `Expect()` invokes the comparison function using `expectedFrame` and `actualFrame` as parameters. The return value of `Expect()` is that of the comparison function. If, however, `ss_receive()` times out before a frame arrives on `fd`, then `Expect()` returns 0 for error.

This concludes our discussion of the library support provided. The C source code for the socket and the header utility libraries has been included in Appendix A. The next 2 chapters will describe how this support can be used to write C programs to perform automated firewall testing.
4. Case Study: AlphaShield

This chapter describes the development and execution of an automated test suite for a simple firewall, the AlphaShield. The test suite uses the send/expect paradigm and the socket library support.

4.1 The AlphaShield

The AlphaShield is a plug and play hardware firewall that sits between the machine to be protected (PC) and the Internet. It has 2 main RJ-45 ports, one connected to the Ethernet port of the PC (PC port) and the other connected to the Internet (Cable/DSL port). The AlphaShield uses stateful inspection to perform packet filtering, with filtering rules that cannot be changed. We will refer to frames that enter the AlphaShield at the Cable/DSL port as inbound frames and those that enter the AlphaShield at the PC port as outbound frames. The basic principle of operation is that the AlphaShield allows all outbound frames (except non-ARP frames that have broadcast destination MAC addresses) but does not allow inbound frames (except ARP frames) unless they are part of an existing session initiated by the PC. This means that if, for example, a host on the Internet tries to ping the PC, the AlphaShield will allow the initial ARP request and reply through but will drop the inbound ICMP request frames. On the other hand, if the PC starts an HTTP session with a web server, the AlphaShield will allow HTTP response frames originating from the web server on the Internet. The AlphaShield does not have a MAC or IP address of its own and does not support features like NAT.

The AlphaShield can operate in one of the following 3 modes, selected using a mode switch [7]:

1. Manual mode: A logical disconnect (all frames except those containing DHCP application layer messages are discarded) occurs after an inactivity period of 15 minutes.
2. Auto mode: The AlphaShield always remains connected. No timeout occurs.

3. Lockout mode: A physical disconnect (all frames are discarded) occurs after an inactivity period of 15 minutes.

A connection can be terminated at any time (either logical or physical disconnect occurs depending on the mode) by pressing the Disconnect button. Connection can be re-established by pressing the Connect button.

4.2 Test Plan

Since there was no documentation available on the AlphaShield's method of operation, trial and error was used to determine the AlphaShield's packet filtering rule set. Based on this rule set, a specification of the AlphaShield (Appendix B) was created by a fellow graduate student, Yong Du. Our test plan is derived from this specification.

4.3 Test Setup

The test setup consists of a Linux PC with 2 Ethernet cards called the driver, which is connected to the AlphaShield. The test setup is shown in Figure 4.1. The first card (eth1) is connected to the PC port of the AlphaShield and the second card (eth2) is connected to the Cable/DSL Modem port. Here eth1 represents the PC and eth2 represents the Internet. Setting the driver's network cards to run in promiscuous mode allows us to route frames to simple destination MAC addresses (e.g. 00:00:00:00:00:02), without the need to determine the actual MAC address of the listening network card. The destination MAC address used should not be the MAC address of the listening network card. This is to prevent the frame received from being sent up the protocol stack for processing. A C application program containing an automated test suite for the AlphaShield runs on the driver.
4.4 Test Cases

Each AlphaShield test case consists of the create, send and expect steps. Outbound frames are sent on eth1 and expected on eth2. Inbound frames are sent on eth2 and expected on eth1. For each test case, we will describe the frame to be sent and indicate whether it should be accepted or dropped.

4.4.1 ARP tests

Both inbound and outbound ARP request and reply frames are allowed through the AlphaShield. The following tests are conducted for both outbound and inbound ARP frames:

- **Broadcast ARP request: accepted.** The ARP request frame has the broadcast destination MAC address.

- **Unicast ARP request: accepted.** The ARP request frame has a non-broadcast destination MAC address.

- **Broadcast ARP reply: accepted.** The ARP reply frame has the broadcast destination MAC address.

- **Unicast ARP reply: accepted.** The ARP reply frame has a non-broadcast destination MAC address.
4.4.2 IP frame tests

We test 3 types of IP frames:

1. Raw IP, i.e., frames which do not have a transport layer header.

2. TCP/IP, except frames that are part of an active FTP session, which will be discussed later.

3. UDP/IP

As described in the specification in Appendix A, the AlphaShield stores state information on a session in the form of a 5-tuple we call the sessionId which is stored in the session table. A sessionId is the tuple \( \langle \text{protocol}, \text{sourceIP}, \text{sourcePort}, \text{destIP}, \text{destPort} \rangle \). When a sessionId is swapped, it becomes the tuple \( \langle \text{protocol}, \text{destIP}, \text{destPort}, \text{sourceIP}, \text{sourcePort} \rangle \). In general, if an outbound frame has a sessionId \( s \), then an inbound frame which is part of the same session will have sessionId \( \text{swap}(s) \).

When the AlphaShield sees an outbound frame, it first compares the frame against the session table entries. If there is no entry corresponding to the outbound frame’s sessionId, then one is created and stored in the session table. When an inbound frame is encountered, the AlphaShield compares a swap of the inbound frame’s sessionId with entries in the session table. If a match is found, the inbound frame is deemed to be part of an existing session initiated by the PC, and is allowed through. If no match is found, the inbound frame is dropped. The sessionId variables and values used in the tests are shown in Figure 4.2. The value of dIP should not be equal to the IP address of the driver’s eth2 interface. sPort and dPort should be ephemeral ports. For raw IP frames, they are both considered to be 0. The test cases are as follows:

- **Outbound frame with broadcast destination MAC address: dropped.** The frame has sessionId \( s \) with broadcast destination MAC address.

- **Inbound frame with broadcast destination MAC address: dropped.** The frame has sessionId \( \text{swap}(s) \) with a broadcast destination MAC address.
\[ s = <P, sIP, sPort, dIP, dPort> \]
\[ s1 = <P1, dIP, dPort, sIP, sPort> \]
\[ s2 = <P, d1IP, dPort, sIP, sPort> \]

where:

- \( P, P1 \) are raw IP, TCP or UDP
- \( sIP = 10.1.1.1 \)
- \( dIP = 10.2.1.2 \)
- \( d1IP = 10.2.1.3 \)
- \( sPort = 0 \) for raw IP, 2000 otherwise
- \( dPort = 0 \) for raw IP, 2001 otherwise

Figure 4.2: SessionId variables and values

- **Stateful Inspection tests**
  
  - *Outbound unicast frame*: accepted. The frame has sessionId s.
  
  - *Inbound unicast frame sessionId swap(s)*: accepted.
  
  - *Inbound unicast frame with sessionId s1*: dropped.
  
  - *Inbound unicast frame with sessionId s2*: dropped.

### 4.4.3 Active FTP

In active FTP, the client connects from an ephemeral port (> 1024) to the FTP server’s command port, port 21. It then informs the server which port it is going to be listening on (p1). The FTP server opens a connection from its data port, port 20 to the client port p1. If the FTP client is behind the AlphaShield, the normal filtering rule set of the AlphaShield will not allow the data connection initiated by the FTP server. This is because the AlphaShield will not allow an inbound frame.
which does not have an entry in its session table. However, the AlphaShield makes an exception for active FTP and allows inbound session initiation if it sees a sessionId entry indicating that the client had previously initiated a session to port 21 of the server. The following test case verifies that the AlphaShield does indeed allow active FTP:

```
sessionId variables
s = (TCP, clientIP, p, serverIP, 21)
s1 = (TCP, serverIP, 20, clientIP, p1)
```

- **Outbound FTP control frame: accepted.** The frame has sessionId s.
- **Inbound FTP data frame: accepted.** The frame has sessionId s1.

### 4.4.4 Session Timeout

Session table entries are not stored forever. For an existing session, if no outbound frame belonging to that session is seen for 5 minutes, the session table entry is removed from the session table. Any inbound traffic belonging to that session will, therefore, no longer be allowed through unless an outbound frame is sent to re-establish the session. This behaviour implies that the session table entries are sessionId, timer pairs. The timer is reset to 0 every time the Alpha Shield sees an outbound frame. Thus, a session can continue for longer than 5 minutes only if the PC issues outbound frames at least once every 5 minutes. To test that the session timeout period is 5 minutes, the following test is carried out:

```
sessionId variables
s = (UDP, sIP, sPort, dIP, dPort)
```

- **Outbound frame: accepted.** The frame has sessionId s.
- **Inbound frame: accepted.** The frame has sessionId swap(s).
- **Pause for 5 minutes**
- **Inbound frame: dropped.** The frame has sessionId swap(s).
4.4.5 Session Table Capacity

The AlphaShield specification states that the session table can hold up to 96 sessionId, timer pairs. When the session table is full and the Alpha Shield sees an outbound frame which needs an entry in the session table, one of the 96 entries in the session table is discarded in favour of the newest entry.

- **97 Outbound frames: all frames accepted.** Each frame has a unique sessionId (UDP, sIP, i, dIP, P) where i varies from 0 to 96.

- **97 Inbound frames: 96 frames accepted, 1 frame dropped.** Each frame has a unique sessionId (UDP, dIP, P, sIP, i) where i varies from 0 to 96.

4.4.6 Denial of Service Attack Tests

A denial of service (DOS) attack is an explicit attempt to prevent legitimate users from using resources [8]. One of the ways to launch a DOS attack is to flood the machine with frames that it has to process, thereby tying up network resources and disrupting legitimate network traffic. The Alpha Shield allows all inbound ARP replies and therefore exposes the PC to being flooded with unsolicited ARP replies. This could cause the PC to drop legitimate ARP replies entries from its ARP cache.

We tested to see if the AlphaShield offered any protection against such DOS attacks. A machine running Microsoft Windows™[9] was used as the target and a Linux machine was the attacker. The AlphaShield’s Cable/DSL port was connected to eth1 of the attacker and its PC port is connected to eth0 of the target. Unsolicited ARP reply frames with all combinations of source MAC addresses ranging from 00:00:00:00:00:00 to 00:00:00:00:FF:FF with corresponding combinations of source IP addresses from 10.1.0.0 to 10.1.255.255 were sent from the attacker to the target. This means that our DOS attack consisted of $256 \times 256 = 65536$ ARP reply frames (each reply with a unique IP-MAC combination would need a new entry in the target’s ARP cache). The destination IP and MAC addresses on the frames are the IP and MAC addresses of eth0 of the target. The frames were constructed using
socketLibrary and headerUtility calls and sent from the Linux machine in a loop with a 100 µsec delay between each frame, producing 6345 frames per second. When the test was in progress, we tried to ping the target from the attacker, but there was no response. This implies that the target’s ARP cache is receiving too many ARP replies to retain the one legitimate ARP entry it needs to send out an ICMP response to the attacker. Hence we can say that the AlphaShield does not offer protection against such DOS attacks.

4.5 Conclusion

The send/expect paradigm can be used effectively to write automated test cases for a straightforward firewall like the AlphaShield. The C application code for the test suite is listed in Appendix C. In the next chapter, we will apply the send/expect paradigm to a more complex firewall: iptables.
5. Case Study: iptables

This chapter describes the use of the send/expect paradigm to create an automated test suite for firewalls built using the Linux firewall administration program, *iptables*. It also describes performance testing to obtain delay characteristics of a Linux router running iptables.

5.1 Netfilter

iptables is built on the Netfilter framework found in Linux 2.4.x kernels. Netfilter provides a raw framework for manipulating packets at the IP network level as they traverse various parts of the kernel [10]. It provides a series of hooks at various points of the protocol stack (Figure 5.1). The following 5 hooks are available:

1. Prerouting: the packet's entry point after having passed simple sanity checks such as the IP checksum being correct. Routing decisions are made after this point.

2. Input: packets destined for a local process are sent here before being passed to the process itself.

3. Forward: packets destined for another interface come to this point.

4. Postrouting: packets leaving the host pass through this hook before being put on the wire.

5. Output: packets created locally pass through this point.

Kernel loadable modules can register functions to listen at any of these hooks [10]. Such a module must specify the priority of the function within the hook. When an IP packet arrives at that point in the protocol stack, the netfilter hook is called from the core networking code; each module that has registered functions for that
hook is invoked in priority order. The module is then given full control over the packet. It can examine/modify packet headers and return one of 5 values which tell netfilter to either:

1. Accept: let the packet continue traversal as usual.
2. Drop: discard the packet.
3. Stolen: forget about the packet; the module has taken over the packet.
4. Queue: queue the packet for user space handling.
5. Repeat: call this hook again.

5.2 iptables Terminology

iptables is a packet selection system consisting of kernel loadable modules as well as user space code.

5.2.1 Rules, Chains, Targets and Policies

A chain is a set of iptables rules. These rules are set from user space using the iptables command. An iptables rule consists of a predicate/action pair. There is
a chain corresponding to each netfilter hook. A packet that has entered a chain is matched against each rule in that chain until a match is found or the end of the chain is reached. If a match is found, i.e., if the packet satisfies the predicate, the action, also called the target of the rule is executed on that packet. The targets correspond to the return values of the netfilter modules. The most commonly used targets are ACCEPT and DROP.

Each chain has a default policy, which is the target applied when the packet reaches the end of the chain without matching any of the rules. Most often, the default policy is set to DROP so that all packets except those that match a rule in the chain are discarded. Also, if a chain does not contain any rules, then the default policy applies to all packets that enter the chain.

5.2.2 Tables
There are 3 independent tables, the filter table, the nat table and the mangle table. Each table has its own built-in chains, e.g., the filter table has input, output and forward chains. Each iptables rule, therefore, must specify the table and the chain within the table that it should be applied to. If no table has been specified, the rule is assumed to apply to a chain within the filter table.

5.3 Iptables Testing
Our automated suite tests iptables rules as applied to the prerouting, forward and postrouting chains. This is because the send/expect paradigm is most useful when the frame is routed through the firewall. We referred to [11] for rule syntax and description.

5.3.1 Test Setup
The test setup (Figure 5.2) consists of 2 Linux machines each with 3 Ethernet cards, eth0, eth1 and eth2. One of the machines is called the driver because it runs our automated test suite and the other machine running iptables is called the system.
under test (SUT). Eth0 of the driver and eth0 of the SUT are connected so as to allow remote connection such as an ssh session between them. Eth1 and eth2 of the SUT are connected to the corresponding interfaces of the driver by using crossover cables. The IP addresses of the different Ethernet interfaces have also been depicted in the figure. Both eth1 interfaces belong to the 10.1.0.0/16 subnet and both eth2 interfaces belong to the 10.2.0.0/16 subnet. The entries in the routing table ensure that a frame entering the SUT on eth1 and having a 10.2.x.x destination IP address is forwarded to eth2 and a frame entering on eth2 with a destination IP address of 10.1.x.x is forwarded to eth1. Static ARP entries are provided on the SUT for all 10.1.x.x and 10.2.x.x destination IP addresses that are used in test frames.

The test suite running on the driver uses the send/expect paradigm. Test cases consist of the following steps:

1. Configure one or more iptables rules remotely on the SUT by establishing an ssh connection on eth0.
2. Create a frame.
3. Send the frame.
4. Expect/Not Expect the frame to return.

A test case sets one or more iptables rules (Step 1). These rules are then tested by creating and sending different frames by repeating steps 2, 3 and 4.

A frame is created and sent out of one of the driver’s Ethernet interfaces, say eth1. The frame’s destination IP address is a 10.2.x.x address which does not match the eth2 IP addresses on either the driver or the SUT. The frame’s destination MAC address is the MAC address of the SUT’s eth1. The driver then waits, expecting/not expecting to see the frame return on its eth2. On the SUT, if the frame passes the iptables rules for the prerouting chain, it is examined and a routing decision is made, i.e., that the frame needs to be sent out through eth2. The destination MAC address of the frame is then overwritten with the entry corresponding to its destination IP
address in the static ARP table of the SUT and the frame enters the forward chain. If the frame passes the iptables rules for the forward chain, it is sent to the postrouting chain. If it passes the iptables rules for the postrouting chain, it leaves the SUT out of eth2. The frame is then received by the driver’s eth2 which is running in promiscuous mode and is passed on to the Expect() function waiting on eth2. If the frame does not pass the iptables rules for the prerouting, forward or postrouting chains on the SUT, it is discarded by the SUT and Expect on the driver does not see the frame come back and eventually times out. The static ARP entries in the SUT ensure that no ARP frames are exchanged between the driver and SUT. This is important because the Send/Expect paradigm works on the assumption that the only frames sent and received on eth1 and eth2 are the ones created and sent out by the driver.
outbound frame = <MAC1,P,sIP,sPort,dIP,dPort>

inbound frame = <MAC2,P,dIP,dPort,sIP,sPort>

where:

P is raw IP, TCP or UDP

MAC1 = 00:00:00:00:00:01

MAC2 = 00:00:00:00:00:02

sIP = 10.1.0.1

dIP = 10.2.0.2

sPort = 2000

dPort = 2001

Figure 5.3: Tuple variables and values

5.3.2 Assumptions

We will denote the frames that enter the SUT on eth1 and leave it on eth2 as outbound frames, and those that enter the SUT on eth2 and leave it on eth1 as inbound frames. Hence, outbound frames have a 10.1.x.x source IP address and a 10.2.x.x destination IP address whereas inbound frames have a 10.2.x.x source IP address and a 10.1.x.x destination IP address. Unless otherwise mentioned, we will assume that the inbound and outbound frame tuples of (Source MAC, Source IP, Source Port, Destination IP, Destination Port) have values as indicated in Figure 5.3.

We use the library functions to initialize the frame and set basic fields such as the IP addresses and ports described above. Special flags or other header fields that require non-zero values for the frame to pass a particular test are handled by modifying the frame after it has been initialized. Also, before each test case is executed, we will ensure that the default iptables policy has been set to DROP, i.e., all frames will be
discarded unless they match the rule set by the test case and that iptables has been flushed, i.e., the rules from previous test cases have been removed.

5.3.3 Basic filter Table Match Operations

The rules applicable to the filter table examine frame header fields such as source and destination IP addresses, source and destination TCP/UDP ports, TCP flags and source MAC address.

5.3.3.1 Input Interface

This rule applies to packets entering the host at a particular interface.

- Set an iptables rule in the forward chain to accept frames entering the SUT on its eth1 interface.

  - *Outbound frame: accepted.*

  - *Inbound frame: dropped.*

  The C application code for this test case is as shown in Figure 5.4.

5.3.3.2 Output Interface

This rule applies to packets leaving the host at a particular interface.

- Set an iptables rule in the forward chain to accept frames leaving the SUT on its eth1 interface.

  - *Outbound frame: dropped.*

  - *Inbound frame: accepted.*

5.3.3.3 Protocol

This rule filters on the IP protocol that the rule applies to. Numeric values found in /etc/protocols can be used. Built-in protocol names that can be used include TCP, UDP, ICMP and ALL.
unsigned char frame[1514];

// set rule to accept frames entering on eth1
system("ssh -1 142.104.101.147 iptables -A FORWARD -i eth1 -j ACCEPT");

// create outbound UDP frame
initUDPEth1(frame);

// send outbound frame -- should get thru
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&UDPCompare);

// create inbound UDP frame
initUDPEth2(frame);

// send inbound UDP frame -- should not get thru
sendFrame(fd2,frame,FRAMELENGTH);
Expect(fd1,frame,NULL);

Figure 5.4: Input Interface test

Accept only UDP frames:

- Set an iptables rule in the forward chain to accept UDP frames.

  - *Outbound UDP frame: accepted.*

  - *Outbound TCP frame: dropped.*

Accept all protocols:

- Set an iptables rule in the forward chain to accept frames having any protocol.

  - *Outbound TCP frame: accepted.*

5.3.3.4 Source IP address

This rule filters on the host or the network source address in the IP header. The IP address can be specified either in the dotted form or as an IP address followed by a mask (to specify a subnet). Here, s1IP = 10.1.0.2.

Accept UDP frames with a certain destination IP:
- Set an iptables rule in the forward chain to accept UDP frames with source IP address sIP.

  - Outbound UDP frame: accepted.

  - Outbound UDP frame with source IP address = s1IP: dropped.

Accept UDP frames originating from a certain subnet:

- Set an iptables rule in the forward chain to accept frames having source IP address 10.1.x.x.

  - Outbound UDP frame having source IP address = s1IP: accepted.

5.3.3.5 Destination IP address

This rule filters on the host or the network destination address in the IP header. The IP address can either be specified in the dotted form or as an IP address followed by a mask (to specify a subnet). Here, d1IP = 10.2.0.3.

Accept UDP frames with a certain destination IP:

- Set an iptables rule in the forward chain to accept UDP frames with destination IP address dIP.

  - Outbound UDP frame: accepted.

  - Outbound UDP frame having destination IP address = d1IP: dropped.

Accept UDP frames destined for a certain subnet:

- Set an iptables rule in the forward chain to accept frames having destination IP address 10.2.0.x.

  - Outbound UDP frame having destination IP address = d1IP: accepted.
5.3.3.6 Fragment
This rule applies to the second and further fragments of a fragmented IP frame.

- Set an iptables rule in the forward chain to accept fragments of an IP frame.
- First fragment of an outbound UDP/IP frame: dropped.
- Last fragment of an outbound UDP/IP frame: accepted.

5.3.4 udp filter Table Match Operations
These UDP header match options can be used only if the protocol specified in the rule is UDP.

5.3.4.1 Source Port
This rule filters on the source port in the UDP header. A single port or a range of ports can be specified.

Accept UDP frames having a certain source port:

- Set an iptables rule in the forward chain to accept UDP frames with source port = sPort

  - Outbound UDP frame: accepted.

  - Outbound UDP frame having source port = dPort: dropped.

Accept UDP frames having a certain range of source ports:

- Set an iptables rule in the forward chain to accept UDP frames having source port ranging from 2000 to 2005.

  - Outbound UDP frame having source port = dPort: accepted.
5.3.4.2 Destination Port

This rule filters based on the destination port in the UDP header. Either a single port or a range of ports can be specified. Here, dPort1 = 2002.

Accept UDP frames having a certain destination port:

- Set an iptables rule in the forward chain to accept UDP frames with destination port = dPort

- **Outbound UDP frame: accepted.**

- **Outbound UDP frame having destination port = dPort1: dropped.**

Accept UDP frames having a certain range of destination ports:

- Set an iptables rule in the forward chain to accept UDP frames having destination port ranging from 2000 to 2005.

- **Outbound UDP frame having destination port = dPort1: accepted.**

5.3.5 tcp filter Table Match Operations

These TCP header match options can be used only if the protocol specified in the rule is TCP. The test cases for TCP source and destination port filters are similar to those for UDP above and will not be discussed.

5.3.5.1 TCP flags

This rule applies a filter based on the flags in the TCP header. Legal values include: SYN, ACK, FIN, RST, URG, PSH, ALL and NONE. Two arguments must be specified: a comma-separated list of the flags to be examined and the list of flags that need to be set for this rule to be applicable to the frame.

- Set an iptables rule in the forward chain to examine the SYN and ACK flags of TCP frames and accept frames with ACK set.

- **Outbound TCP frame with ACK set: accepted.**
- Outbound TCP frame with ACK reset: dropped.

The above test case was repeated for all the other TCP flags.

### 5.3.5.2 SYN

This rule is a special case of the TCP flags filter being applied to a TCP frame whose SYN flag is set and whose ACK and FIN flags are reset. This frame is important because it is used initiate TCP connection.

- Set an iptables rule in the forward chain to accept TCP frames with SYN set and ACK, FIN reset.

- Outbound TCP frame with SYN set and ACK, FIN reset: accepted.

- Outbound TCP frame with SYN, ACK set and FIN reset: dropped.

### 5.3.6 mac filter Table Match Extension

This extension of the filter table requires the mac module to be loaded before the match can be specified. The source MAC address in the Ethernet header is the only match possible.

- Set an iptables rule in the forward chain to accept frames with source MAC address 00:00:00:00:00:01

- Outbound UDP frame: accepted.

- Inbound UDP frame: dropped.

### 5.3.7 multiport filter Table Match Extension

This extension of the filter table requires the multiport module to be loaded. It allows filtering based on comma-separated lists of source or destination ports for TCP and UDP frames. Here, port1 = 2002.

Accept UDP frames having certain source ports:
• Set an iptables rule in the forward chain to accept UDP frames with source port 2000 or 2002

• *Outbound UDP: accepted.*

• *Outbound UDP frame having source port port1: accepted.*

• *Outbound UDP frame having source port dPort: dropped.*

Accept UDP frames having certain destination ports:

• Set an iptables rule in the forward chain to accept UDP frames with destination port 2001 or 2002

• *Outbound UDP frame: accepted.*

• *Outbound UDP frame having destination port port1: accepted.*

• *Outbound UDP frame having destination port sPort: dropped.*

5.3.8 *limit filter Table Match Extension*

This extension requires the limit module to be loaded. Token bucket like filtering is provided, i.e., the maximum bucket size can be specified, after which the token rate can be limited to a desired value. Token rates can be specified in terms of matches per second, per minute, per hour or per day.

1. Set an iptables rule in the forward chain to accept a burst of 3 outbound UDP frames and limit to 1 match per second thereafter.

2. *Burst of 3 outbound UDP frames: all frames accepted.*

3. *Outbound UDP frame: dropped.*

4. *Pause for 3 seconds*

5. *Burst of 3 outbound UDP frames: all frames accepted.*
6. *Pause for 1 second*

7. *Outbound UDP frame: accepted.*

An initial burst of 3 frames that match the rule (Step 2) are all accepted. Immediately following the burst, a fourth frame is sent (Step 3). This is dropped because the token bucket is empty. After a pause of 3 seconds (Step 4), during which the token bucket accumulates 3 tokens (one per second), another burst of 3 frames that match the rule are sent and are all accepted (Step 5). This burst empties the bucket. A further 1 second pause (Step 6) allows the bucket to collect one token. This allows the next outbound UDP frame to be accepted (Step 7).

5.3.9 **state filter Table Match Extension**

The *state* extension module adds connection tracking capabilities to iptables. The state of the ongoing TCP or UDP connection can be matched. The states that can be specified are:

1. New: refers to the first frame in a connection, e.g., a TCP SYN frame or the first UDP frame in a session.

2. Established: refers to frames that are part of an ongoing connection, e.g., the second and third frames during TCP connection establishment, a TCP frame with ACK sent after the 3 way connection establishment has occurred or subsequent UDP datagrams exchanged between the same ports of the 2 hosts.

3. Related: refers to frames that are starting a new connection which is associated with an existing connection, e.g., a FTP data transfer.

4. Invalid: refers to a frame that is not associated with any known session, e.g., a TCP frame with both SYN and ACK reset.

5. Assured: refers to a connection that has seen traffic in both directions after connection has been established. An assured connection will be preferentially
retained in the conntrack table under heavy load, i.e., when the conntrack table
contains the maximum possible number of tracked connections. In these cir-
cumstances, non-assured connections will be erased from the conntrack table.

Internally, every frame that starts a new connection causes an entry in the
conntrack table. As in the AlphaShield, the 5 tuple consisting of the IP protocol,
e.g., TCP, UDP, or ICMP, source IP address, source port, destination IP address and
destination port are stored for each connection. Two sets of 5 tuples are saved for each
connection: one applies to the direction of the frame that initiated the connection and
the other is for frames that are expected in the opposite direction. In general, these
two 5 tuples are swaps of each other, i.e., if one of them is \((p,sIP,sPort,dIP,dPort)\),
then the other is \((p,dIP,dPort,sIP,sPort)\). Each conntrack entry also contains flags
indicating the current status of the connection. Conntrack table entries are purged
when there has been no traffic on the connection for a certain period of time which
we will refer to as the timeout period, or if the connection is explicitly terminated as
in TCP.

5.3.9.1 UDP connection tracking

The conntrack entry for a UDP session can be in the new (unreplied), estab-
lished or assured state. The first outbound frame causes the entry to be created and
marked as an unreplied connection. A swapped inbound frame will change the state
to established, with a timeout of 30 seconds. Any subsequent inbound or outbound
frame will change the state from established to assured with a timeout of 180 seconds.

Session Id variables

\[
\begin{align*}
s_1 &= (UDP, 10.2.0.2, 2001, 10.1.0.1, 2002) \\
s_2 &= (TCP, 10.2.0.2, 2001, 10.1.0.1, 2000)
\end{align*}
\]

- Set iptables rules in the forward chain to accept outbound UDP frames belong-
ing to new and established connections and inbound UDP frames that are part
of an established connection.
- **Outbound UDP frame: accepted.** The frame creates an entry in the conntrack table marked as an unreplied connection, with a timeout of 30 seconds.

- **Pause for 25 seconds**

- **Inbound UDP frame: accepted.** The conntrack table entry is marked as an established connection and the timer is reset to 0.

- **Inbound UDP frame with different destination port: dropped.** The frame has session Id s1.

- **Inbound TCP frame: dropped.** The frame has session Id s2.

- **Pause for 35 seconds**

- **Inbound UDP frame: dropped.** The frame is dropped because timeout has occurred.

- **Assured Connection**
  - **Outbound UDP frame: accepted.** The conntrack entry is marked as an unreplied connection.
  
  - **Inbound UDP frame: accepted.** The conntrack entry is marked as an established connection.
  
  - **Outbound UDP frame: accepted.** The conntrack entry is marked as an assured connection. The timeout period changes to 180 seconds.
  
  - **Pause for 175 seconds**
  
  - **Inbound UDP frame: accepted.** Timeout has not yet occurred and the timer is reset to 0
  
  - **Pause for 185 seconds**
  
  - **Inbound UDP frame: dropped.** Timeout has occurred.
5.3.9.2 TCP Connection tracking

TCP connection tracking follows the TCP state transition diagram [12]. Hence the TCP connection goes through the SYN_SENT and SYN_RECEIVED states during connection establishment and the FIN_WAIT, CLOSE_WAIT and TIME_WAIT states during connection termination. Each state has its own timeout period.

Session Id variables

\[ s_1 = (\text{TCP}, 10.2.0.2, 2001, 10.1.0.1, 2002) \]
\[ s_2 = (\text{UDP}, 10.2.0.2, 2001, 10.1.0.1, 2000) \]

- Set iptables rules in the forward chain to accept outbound TCP frames belonging to new and established connections and inbound TCP frames that are part of an established connection.

- Outbound SYN TCP frame: accepted. The frame has SYN set and ACK reset. It creates an entry in the conntrack table marked with the SYN_SENT flag having a timeout period of 120 seconds.

- Inbound SYN/ACK TCP frame: accepted. The frame has SYN and ACK set and acknowledgement number a. The conntrack table entry is marked with the SYN_RECEIVED with a timeout period of 60 seconds.

- Outbound ACK TCP frame: accepted. The frame has ACK set and SYN reset with sequence number a. The conntrack entry is marked established and assured with a timeout period of 5 days.

- Inbound TCP frame: accepted. The frame has ACK set and SYN reset.

- Inbound TCP frame with different destination port: dropped. The frame has session Id s1.

- Inbound UDP frame: dropped. The frame has session Id s2.

In the case of TCP, connection termination occurs when:
1. the connection is explicitly shut down because one of the hosts resets it or
2. the connection is terminated normally.

For the following tests, we will assume that the iptables rules are the same as
above and that a TCP connection has already been established for TCP frames with
session IDs as described in Figure 5.3.

- **Abnormal Termination**
  - *Outbound TCP frame: accepted.* The frame has RST set. The entry in the
    conntrack table is marked CLOSE with a timeout period of 10 seconds.
  - *Pause for 5 seconds*
  - *Inbound TCP frame: accepted.* The frame has ACK set and SYN reset.
    Timeout has not yet occurred. The timer is reset to 0.
  - *Pause for 15 seconds*
  - *Inbound TCP frame: dropped.* The frame has ACK set and SYN reset.
    Timeout has occurred and the connection has been terminated.

- **Normal Termination initiated by outbound FIN**
  - *Outbound TCP frame: accepted.* The frame has FIN and ACK set. The
    entry in the conntrack table is marked FIN_WAIT with a timeout period
    of 120 seconds.
  - *Inbound TCP frame: accepted.* The frame has ACK set.
  - *Inbound TCP frame: accepted.* The frame has FIN and ACK set. The
    entry in the conntrack table is marked TIME_WAIT with a timeout period
    of 120 seconds.
  - *Outbound TCP frame: accepted.* The frame has ACK set.
  - *Pause for 115 seconds*
Inbound TCP frame: accepted. The frame has ACK set and SYN reset.
Timeout has not yet occurred. The timer is reset to 0.

Pause for 125 seconds

Inbound TCP frame: dropped. The frame has ACK set and SYN reset.
Timeout has occurred and the connection has been terminated.

Normal Termination initiated by inbound FIN

Inbound TCP frame: accepted. The frame has FIN and ACK set. The entry in the conntrack table is marked CLOSE-WAIT with a timeout period of 60 seconds.

Outbound TCP frame: accepted. The frame has ACK set.

Outbound TCP frame: accepted. The frame has FIN and ACK set. The entry in the conntrack table is marked TIME-WAIT with a timeout period of 120 seconds.

Inbound TCP frame: accepted. The frame has ACK set.

Pause for 115 seconds

Inbound TCP frame: accepted. The frame has ACK set and SYN reset.
Timeout has not yet occurred. The timer is reset to 0.

Pause for 125 seconds

Inbound TCP frame: dropped. The frame has ACK set and SYN reset.
Timeout has occurred and the connection has been terminated.

5.3.10 NAT Table Target Extensions

Rules applied to the NAT table modify either the source/destination IP address or port of the packet. We will discuss tests for 3 types of NAT: Source NAT (SNAT), Masquerade, and Destination NAT (DNAT).

SNAT is used when the hosts in a private network share a single public IP address. Each host has a unique IP address by which it is identified on the private
network. However, this private IP address is not known on the Internet. A NAT-enabled device sits between the private network and the Internet. It translates the source IP address of an outbound frame to the public IP address. Correspondingly, it translates the destination IP address of an inbound frame to the private IP address of the host that initiated the connection. Thus the inbound frame gets forwarded to the correct host. In addition to source IP address translation, source port translation can also be carried out.

Masquerade is a special type of SNAT in which the source IP address of the frame is replaced with the IP address of the outgoing interface. It is preferred over SNAT when the NAT-enabled device has a dynamic IP address assigned by dial-up or DHCP connections. Source port translation can also be carried out.

DNAT is used when NAT-enabled device routes packets to servers on an internal network based on the destination port of the packet. Again, the servers all share a public IP address. Both destination IP address and port can be translated.

Frames that are translated because they match a NAT table rule have a conntrack table entry. Unlike the state table conntrack entries however, the 2 tuples saved for each NAT entry are not exact swaps of each other. This is because of the changes in source or destination IP addresses or ports caused by NAT.

5.3.10.1 SNAT nat Table Target Extension

SNAT is done after the routing decision has been made and immediately before sending the packet out. Thus the SNAT target is legal only in the postrouting chain. Either the source IP address or the source port or both can be translated. A range of IP addresses and/or ports can be specified.

Session Id variables

\[ s = (UDP, 10.1.0.1, 2000, 10.2.0.2, 2001) \]
\[ s1 = (UDP, 10.1.0.2, 2000, 10.2.0.2, 2001) \]
\[ s2 = (UDP, 10.2.0.2, 2001, 10.2.0.3, 2000) \]
\[ s3 = (UDP, 10.2.0.2, 2001, 10.2.0.4, 15000) \]
\[ s_4 = \langle \text{UDP}, 10.2.0.2, 2001, 10.2.0.5, 15000 \rangle \]

Translate the source IP address of outbound frames:

- Set an iptables rule in the postrouting chain to translate the source IP address of outbound frames to 10.2.0.3.

- Outbound UDP frame: accepted. The frame has session Id \( s \), which is translated to swap(s2) just before it leaves on the SUT's eth2.

- Inbound UDP frame with session Id \( s_2 \): accepted. The frame's session Id is translated from \( s_2 \) to swap(s) as soon as it enters on the SUT's eth2.

Translate the source IP address and the source port of outbound frames:

- Set an iptables rule in the postrouting chain to translate the source IP address of outbound frames to 10.2.0.4 or 10.2.0.5 and source port to 15000.

- Outbound UDP frames with session Ids \( s \) and \( s_1 \): both accepted. One frame's session Id translated to swap(s3) and the other's to swap(s4).

- Inbound UDP frames with session Ids \( s_3 \) and \( s_4 \): both accepted. One frame's session Id translated to swap(s) and the other's to swap(s1).

5.3.10.2 MASQUERADE nat Table Target Extension

Like SNAT, masquerade is carried out in the postrouting chain. Source port translation can be carried out by specifying a single port or a range of ports.

Session Id variables

\[ s = \langle \text{UDP}, 10.1.0.1, 2000, 10.2.0.2, 2001 \rangle \]
\[ s_1 = \langle \text{UDP}, 10.1.0.2, 2000, 10.2.0.2, 2001 \rangle \]
\[ s_2 = \langle \text{UDP}, 10.2.0.2, 2001, 10.2.2.2, 2000 \rangle \]
\[ s_3 = \langle \text{UDP}, 10.2.0.2, 2001, 10.2.2.2, 17000 \rangle \]
\[ s_4 = \langle \text{UDP}, 10.2.0.2, 2001, 10.2.2.2, 17001 \rangle \]

Masquerade the source IP address of outbound frames:
Set an iptables rule in the postrouting chain to masquerade outbound frames.

- **Outbound UDP frame: accepted.** The frame has session Id s, which gets translated to swap(s2) just before it leaves on the SUT's eth2.

- **Inbound UDP frame with session Id s2: accepted.** The frame’s session Id is translated to swap(s) as soon as it enters on the SUT’s eth2.

Masquerade the source IP address and source port of outbound frames:

- Set an iptables rule in the postrouting chain to translate the source port to 17000 or 17001.

- **Outbound UDP frames with session Ids s and s1: both accepted.** One frame’s session Id translated to swap(s3) and the other’s to swap(s4).

- **Inbound UDP frames with session Ids s3 and s4: both accepted.** One frame’s session Id translated to swap(s) and the other’s to swap(s1).

### 5.3.10.3 DNAT nat Table Target Extension

DNAT is done just before the routing decision is made. It is legal in the prerouting and output chains. Either the destination IP address or the destination port or both can be translated. A range of IP addresses and/or ports can be specified.

**Session Id variables**

\[
\begin{align*}
    s &= (\text{UDP, 10.2.0.2, 2001, 10.1.0.1, 2000}) \\
    s_1 &= (\text{UDP, 10.2.0.2, 2001, 10.1.0.2, 2000}) \\
    s_2 &= (\text{UDP, 10.1.0.3, 2000, 10.2.0.2, 2001}) \\
    s_3 &= (\text{UDP, 10.1.0.4, 15000, 10.2.0.2, 2001}) \\
    s_4 &= (\text{UDP, 10.1.0.5, 15000, 10.2.0.2, 2001})
\end{align*}
\]

Translate the destination IP address of inbound frames:

- Set an iptables rule in the prerouting chain to translate the destination IP address of inbound UDP frames having destination port 2000 to 10.1.0.3.
• **Inbound UDP frame: accepted.** The frame has session Id s, which is translated to swap(s2) just before it is routed from the SUT's eth2 to its eth1.

• **Outbound UDP frame with session Id s2: accepted.** The frame's session Id is translated to swap(s) just after it is routed from the SUT's eth1 to its eth2.

Translate the destination IP address and the destination port of inbound frames:

• Set an iptables rule in the prerouting chain to translate the destination IP address of UDP frames having destination port 2000 to 10.1.0.4 or 10.1.0.5 and the destination port to 15000.

• **Inbound UDP frames with session Ids s and s1: both accepted.** One frame's session Id is translated to swap(s3) and the other's to swap(s4).

• **Outbound UDP frames with session Ids s3 and s4: both accepted.** One frame's session Id is translated to swap(s) and the other's to swap(s1).

### 5.4 Performance Testing

The raw socket library can be used to carry out performance testing on a Linux router running iptables. The objective is to measure round trip delay as a function of framelength. We used framelengths of 64, 128, 256, 512, 1024, 1280 and 1518 bytes as recommended by [13]. The test setup and configuration is as in Figure 5.2. We will call this configuration A. No iptables rules are set and the default policy for the forward chain is ACCEPT. The experiment involves the following steps:

1. Create a UDP frame $f_1$, having a 10.1.x.x source IP address and 10.2.x.x destination IP address and the current timestamp as a timeval structure ($t_1$) occupying its last 8 bytes.

2. Send $f_1$ on eth1 of the driver.

3. Receive $f_1$ back on eth2 of the driver after it has been routed through the SUT.
4. Extract \( t_1 \) from \( f_1 \) and calculate the difference between the current time and \( t_1 \). This is the total roundtrip delay of \( f_1 \).

The steps outlined above are repeated 10 times for each of the 7 framelongths and the average round trip time for that framelongth is determined. However, the round trip time calculated above includes: 1) the time elapsed in the driver before the frame is put on the wire at \( \text{eth1} \) and 2) processing time in the period after the frame picked up by the driver's network card and before it is handed off to the waiting raw socket on \( \text{eth2} \). To calculate the delay introduced by the driver itself, \( \text{eth1} \) and \( \text{eth2} \) on the driver are connected using a crossover cable (configuration B) and the same steps detailed above are carried out for each of the 7 framelongths. The plots of framelongth versus round trip delay for configurations A and B conducted at link rates of 10 Mbps and 100 Mbps are almost linear: correlation coefficients are very close to 1. Thus \( aN + b \) is the round trip delay where \( N \) is the framelongth in bytes, can be used to represent the relationship. Here \( a \) represents the time needed to transmit bits at a particular link rate. Since \( N \) is specified in bytes, \( a \approx 2 \times 8/\text{LinkRate} \) for configuration A and \( a \approx 8/\text{LinkRate} \) for configuration B. The bits have to be clocked twice for configuration A, since the Linux router behaves as a store and forward device and waits to receive all the bits on \( \text{eth1} \) before making a routing decision. At 10 Mbps, \( b \approx 155 \mu \text{seconds} \) for configuration A and \( b \approx 188 \mu \text{seconds} \) for configuration B. At 100 Mbps, \( b \approx 123 \mu \text{seconds} \) for configuration A and \( b \approx 165 \mu \text{seconds} \) for configuration B. Thus, for a particular link rate, the value of \( b \) is higher for configuration B than for configuration A. This difference represents the time needed for routing and iptables filtering. Plots of delay vs. framelongth at 10 Mbps and 100 Mbps are shown in Figure 5.5 and Figure 5.6 respectively. The plots indicate a linear relationship between round trip delay and framelongth for both 10 and 100 Mbps working with configurations A and B.
Figure 5.5: Performance Testing at 10 Mbps

Figure 5.6: Performance Testing at 100 Mbps
5.5 Conclusions

Automated iptables correctness testing using the send/expect paradigm is convenient for the prerouting, forward and postrouting chains. Most of the rules applicable to the input and output chains are also applicable to one or more of the prerouting, forward and postrouting chains. Hence there is no significant loss in test coverage of important iptables rules. The performance testing using the library support provides useful delay characteristics for a Linux router running iptables. It can be easily extended to measure performance in the presence of iptables rules so as to provide a better picture of delay resulting from large rule sets.
6. Related Work

Because of the time and money spent on testing, software test automation has become an important research area. Among the approaches to automated testing are [14]:

- Automatically generate test inputs based on analysis of a given implementation.
- Automatically generate test inputs and expected outputs from a formal specification.
- Develop algorithms to automatically generate test inputs and expected outputs based on the tester’s knowledge of a specification or an implementation.

The first approach was used in early work by Clarke [15] and more recently, in works by Bertolino and Marre [16] and Gotlieb et al. [17]. Although this approach can achieve a certain amount of structural test coverage, it has limited application because output checking is either ignored or too difficult.

The second approach was seen in early work by Bouge et al. [18] and Gannon et al. [19]. More recent work that uses this approach can be found in [20] and [21]. This method requires formal specifications which are seldom available in industry.

The third approach performs output checking without the need for a formal specification. This approach relies on the tester’s knowledge of the system and can therefore result in some loss of test coverage. It has, however, been demonstrated to have useful applications. This approach is seen in work by Daley [14] that develops the Roast framework for automated testing of Java classes.

Automation of test cases is easier when product line testing, as opposed to product testing, is carried out. Commonalities and variabilities among products in a product line are identified and frameworks that implement the commonalities are created. The variabilities are viewed as parameters. Research on the development of frameworks to support software product lines can be found in [22, 23, 24].
Works on firewall testing can be divided into 2 main categories: 1) testing firewall product lines such as the netfilter/iptables framework and 2) testing specific firewalls based on the firewall product, e.g., testing a script that configures a particular iptables-based firewall.

A formal methodology for testing firewalls is described in [3]. This work is motivated by the fact that field testing of firewalls is being carried out using simple checklists of known vulnerabilities without taking into account the firewall's target network topology and operational environment. The paper proposes a testing methodology based on creating a connected hypergraph model of the firewall's network which is used to prove formally that the topology of the network specifies the sufficient conditions for protection against attacks. Test cases can then be deduced from this specification.

Classification of firewall vulnerabilities forms the basis of the work in [25]. A firewall vulnerability is defined as an error in firewall design, implementation or configuration, which can be exploited to attack the network being protected. Firewall internals are examined and matrices of firewall operations (like IP address/port filtering) versus vulnerability causes (like boundary checking) are created. The tester can then use this information to determine the test cases required. Also, by creating these matrices for different firewalls, common problems can be identified; this can help create automated firewall test suites.

Another approach to firewall testing is taken in the firewall test suite in [26]. The firewall tester (ftester) is a tool designed for testing firewall filtering policies. It consists of a group of Perl scripts: a packet generator script (ftest) and a sniffer (ftestd). These scripts are run on 2 different machines separated by the firewall. ftest injects marked packets (entries in a configuration file) whose data part contains a custom signature. The sniffer intercepts these marked packets. Each script writes to a log file. When a log file comparison utility (freport) is run on the 2 log files, it reports which frames did not reach the sniffer. From this information, the filtering rule set of the firewall can be determined. On the other hand, if the test cases were
crafted so that they tested a known firewall rule set, the output from freport would indicate correctness testing results for that rule set.

The official iptables test suite created by Rusty Russell, a netfilter core team member, is an excellent example of a completely automated firewall test suite [27]. It consists of a group of shell scripts that have a send/expect flavour. Post 2.2 releases of the Linux kernel contain a kernel loadable module called ethertap which can create virtual Ethernet interfaces. The iptables test script uses these ethertap interfaces in much the same way as we use eth1 and eth2. Rules applicable to the prerouting, forward and postrouting chains are tested by routing Ethernet frames from one ethertap interface to the other. Frames received are written to a log file, whose contents are displayed in case of error. Errors are reported immediately after the test case has executed. Iptables internals are also tested, e.g., the state module is tested by constructing conntrack entries and searching the conntrack table for them.

Commercial firewall testing tools such as Firewall Informer [28] run in a Microsoft Windows™ environment. A free Linux based firewall test tool is Dr. Morena [29]. Not much technical information is available on these tools. Both tools do, however, use the concept of 2 Ethernet interfaces separated by a firewall to send/receive frames.
7. Conclusions

Firewalls are the probably the most popular internet security products today. Most firewall testing, however, is still carried out using port scanners and other forms of network reconnaissance. Even though formal firewall testing methodologies have not evolved as much as the firewalls themselves, the sheer number of new firewall products flooding the market has forced researchers and corporations alike to turn their attention towards developing firewall test strategies and tools. Effective firewall testing strategies will, in addition to specifying much needed firewall evaluation criteria, result in early detection of defects, reduced time-to-market and more reliable products.

7.1 Contributions

We consider firewalls to be a product line. Each firewall has at least 2 ports and works according to a set of filtering rules. Our framework for firewall testing takes advantage of these commonalities. It offers the firewall test analyst a straightforward interface which can be used to quickly and conveniently implement test cases for the specific firewall under test.

The socket library hides the complexity involved in transmitting and receiving Ethernet frames using Linux packet sockets. The header utility library reduces the creation of IP frames to a few simple function calls. The comparison functions can be easily modified to compare as many frame header fields as required. In addition, a test suite created using our framework is completely automated and involves no log file creation or comparison; the outcome of a correctness test case is known immediately after it has executed.

A wide range of firewall features such as stateful filtering and NAT can be tested. The AlphaShield and iptables test suites demonstrate the flexibility of the framework; it can be used to test both software and hardware firewalls. The iptables performance
testing demonstrates that the framework can be used effectively to conduct firewall performance testing.

A research paper titled “Testing iptables”, which draws from the work in this thesis, has been accepted to CASCON 2003.

7.2 Future Work

One of the drawbacks of the send/expect paradigm is that the frames transmitted need to be routed through the firewall. In other words, the paradigm will have to be extended before it can be used to test frames destined for the firewall host itself.

Also, the frames currently created using the library support do not contain correct TCP and UDP checksums. Therefore, at this time, the framework can be used only to test firewalls that operate at the network layer. Adding support for transport layer checksum calculations or the use of transport layer sockets for frame transmission and/or reception would, therefore be useful.

Further development of the library should include adding support for creating frames with transport layer protocols other than TCP and UDP. ICMP is one such protocol. Also, the creation of a graphical user interface to set frame headers and data would make the framework useful to less technical testers.
Bibliography


[29] Dr. Morena.

APPENDIX A
Library Support Code

A.0.1 socketLibrary.h

 ifndef SOCKETLIBRARY_H
 define SOCKETLIBRARY_H

 // sourceMAC + destMAC + type + CRC
 define SS EthHDRLEN 18

 // initialize an ethhdr structure
 // preconditions
 // *e is an allocated ethhdr structure
 void ss_setEthernetHeader
 (struct ethhdr* e,
  const unsigned char* dstMAC,const unsigned char* sourceMAC, unsigned short p);

 // open a raw socket with protocol ETH_P_ALL and returns its file descriptor
 // preconditions:
 int ss_socket();

 // bind raw socket fd to ethernet interface i
 // preconditions
 // fd is a open file descriptor returned by ss_socket
 // i is of the form ethN
 void ss_bind(int fd,char* i);

 // send a raw ethernet frame
 // preconditions
 // fd is a open file descriptor returned by ss_socket
 // buf points to at least len bytes where len in [64..1514]
 void ss_send
 (int fd,unsigned char* buf,int len);
// receive next ethernet frame and return number of bytes read or -1 on error
// ethernet header in e
// data portion contained in d
// note: because ss_socket uses ETH_P_ALL frames of any protocol are captured
// preconditions
// fd is a open file descriptor returned by ss_socket
// *e is an allocated ethhdr structure
// d points to at least 1500 bytes
// t is the timeout in seconds; t=-1 makes ss_receive block forever
int ss_receive(int fd,struct ethhdr *e,unsigned char* d,int t);

// close socket fd
// preconditions: fd is a open file descriptor returned by ss_socket
void ss_close(int fd);

#endif

A.0.2 socketLibrary.c

#include<errno.h>
#include<sys/socket.h>
#include<linux/if_ether.h>
#include<linux/if_packet.h>
#include<netinet/if_ether.h>
#include<sys/ioctl.h>
#include<net/if.h>
#include<sys/time.h>
#include "socketLibrary.h"

static unsigned char buf[1514];

void ss_setEthernetHeader
(struct ethhdr* e,
 const unsigned char* dstMAC,const unsigned char* sourceMAC,unsigned short p) {
    memcpy(e->h_dest,dstMAC,6);
memcpy(e->h_source,sourceMAC,6);
e->h_proto = htons(p);
}

int ss_socket() {
    int fd;
    if ((fd = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) < 0)
        perror("Error creating packet socket");
    return(fd);
}

// return sockaddr_ll whose ifindex = i's index and family = AF_PACKET
struct sockaddr_ll getLinkLayerAddress(int fd, char* i) {
    struct ifreq ifr;
    struct sockaddr_ll interfaceAddr;

    memset(&interfaceAddr, 0, sizeof(interfaceAddr));
    memset(&ifr, 0, sizeof(ifr));

    memcpy(&ifr.ifr_name, i, IFNAMSIZ);
    ioctl(fd, SIOCGLIFINDEX, &ifr);

    interfaceAddr.sll_ifindex = ifr.ifr_ifindex;
    interfaceAddr.sll_family = AF_PACKET;
    return(interfaceAddr);
}

void setPromiscuousMode(int fd, char* i) {
    struct ifreq ifr;
    struct packet_mreq mreq;

    memset(&ifr, 0, sizeof(ifr));
    memcpy(&ifr.ifr_name, i, IFNAMSIZ);
memset(&mreq, 0, sizeof(mreq));
mreq.mr_ifindex = ifr.ifr_ifindex;
mreq.mr_type = PACKET_MR_PROMISC;
mreq.mr_alen = 6;

setsockopt(fd, SOL_PACKET, PACKET_ADD_MEMBERSHIP, (void*)kmreq, (socklen_t)sizeof(mreq));
}

void ss_bind(int fd, char* i) {
    struct sockaddr_ll interfaceAddr = getLinkLayerAddress(fd, i);

    if (bind(fd, (struct sockaddr*) &interfaceAddr, sizeof(interfaceAddr)) < 0)
        perror("Error binding socket");
    setPromiscuousMode(fd, i);
}

void ss_send(int fd, unsigned char* buf, int len) {
    if (send(fd, buf, len, 0) < 1)
        perror("Error while sending");
}

int ss_receive(int fd, struct ethhdr *e, unsigned char* d, int t) {
    int n = 0;
    fd_set fdRead;
    int s;
    struct timeval tv;

    if (t == -1) {
        if ((n = read(fd, buf, 1514)) < 0)
            perror("Error while receiving");
    }
} 
else {
    FD_ZERO(&fdRead);
    FD_SET(fd, &fdRead);
    tv.tv_usec = 0;
    tv.tv_sec = t;

    if ((s = select(fd+1,&fdRead,NULL,NULL,&tv)) < 0)
        perror("Error in select\n");
else {
    if (s>0) {
        if (FD_ISSET(fd,&fdRead)) {
            if ((n = read(fd,buf,1514)) < 0)
                perror("Error while receiving");
        else {
            memcpy(e,buf,14);
            memcpy(d,buf+14,n-14);
        }
    }
}
}
return(n);

void ss_close(int fd) {
    close(fd);
}

A.0.3  headerUtility.h

#ifndef HEADERUTILITY_H
#define HEADERUTILITY_H

//length of IP header excluding options
#define IPHDRLEN 20
struct ethIP_arphdr {
    struct arphdr arphdr;
    unsigned char ar_sha[6];  // Sender hardware address
    unsigned char ar_sip[4];  // Sender IP address
    unsigned char ar_tha[6];  // Target hardware address
    unsigned char ar_tip[4];  // Target IP address
};

// calculate the IP checksum and assign it to ip->check
// preconditions
//    *ip is an allocated iphdr structure
//    *ip does not contain options
void setIPCksum(struct iphdr* ip);

//initialize an iphdr struct
//preconditions
//    *ip is an allocated iphdr struct
//    *ip does not contain options
void setIPHeader
(struct iphdr* ip,const unsigned char* destIP,const unsigned char* sourceIP,
     unsigned short protocol,unsigned short framelength);

//initialize a udphdr struct
//preconditions
//    *udp is an allocated udphdr struct
void setUDPHeader
(struct udphdr* udp,unsigned short destPort,unsigned short sourcePort,
     unsigned short framelength);

//initialize a tcphdr struct
//preconditions
//    *tcp is an allocated tcphdr struct
void setTCPHeader
(struct tcphdr* tcp,unsigned short destPort,unsigned short sourcePort);

//initialize a ethIP_arphdr struct
//preconditions
// *arp is an allocated ethIP_arphdr struct
void setARPHeader
(struct ethIP_arphdr* arp,unsigned char* senderMAC,unsigned char* senderIP,
 unsigned char* targetIP);

// extract an iphdr struct
// returns pointer to first byte of IP payload
// preconditions
// *ip is an allocated iphdr structure
// *buf is the first byte of an ip packet
unsigned char* getIPHeader(struct iphdr* ip,unsigned char* buf);

// extract a udphdr struct
// returns pointer to first byte of UDP payload
// preconditions
// *udp is an allocated udphdr structure
// *buf is the first byte of an udp packet
unsigned char* getUDPHeader(struct udphdr* udp,unsigned char* buf);

// extract a tcphdr struct
// returns pointer to first byte of TCP payload
// preconditions
// *tcp is an allocated tcphdr structure
// *buf is the first byte of an tcp packet
unsigned char* getTCPHeader(struct tcphdr* tcp,unsigned char* buf);

// extract a ethIP_arphdr struct
// preconditions
// *arp is an allocated ethIP_arphdr structure
// *buf is the first byte of an arp packet
void getARPHeader(struct ethIP_arphdr* arp, unsigned char* buf);

#endif

A.0.4  headerUtility.c
#include<netinet/ip.h>
#include<netinet/udp.h>
#include<netinet/tcp.h>
#include "headerUtility.h"

static unsigned char buf[1514];

void setIPCksum(struct iphdr* ip) {
    unsigned short word16;
    unsigned int sum=0;
    unsigned short i;
    unsigned char* buff;

    ip->check = 0;
    buff = (unsigned char*) ip;

    for (i = 0; i < 20; i += 2){
        word16 = ((buff[i]<<8)&0xFF00) + (buff[i+1]&0xFF);
        sum += (unsigned int) word16;
    }

    while (sum>>16)
        sum = (sum & 0xFFFF)+(sum >> 16);

    sum = ~sum;
    ip->check = htons((unsigned short)sum);
}

void setIPHeader
(struct iphdr* ip, const unsigned char* destIP, const unsigned char* sourceIP,
unsigned short protocol, unsigned short framelen) {
    memset(ip, 0, sizeof(struct iphdr));
    ip->version = 4;
    ip->ihl = 5;
    ip->tos = 0;
    ip->tot_len = htons(framelen);
    ip->id = 0;
    ip->frag_off = 0;
    ip->ttl = IPDEFTTL;
    ip->protocol = protocol;
    memcpy(&(ip->saddr), sourceIP, 4);
    memcpy(&(ip->daddr), destIP, 4);
    setIPCbsum(ip);
}

void setUDPHeader (struct udphdr* udp, unsigned short destPort, unsigned short sourcePort,
                     unsigned short framelen) {
    memset(udp, 0, sizeof(struct udphdr));
    udp->source = htons(sourcePort);
    udp->dest = htons(destPort);
    udp->len = htons(framelen);
}

void setTCPHeader (struct tcphdr* tcp, unsigned short destPort, unsigned short sourcePort) {
    memset(tcp, 0, sizeof(struct tcphdr));
    tcp->source = htons(sourcePort);
    tcp->dest = htons(destPort);
    tcp->window = htons(17244);
}

void setARPHdr (struct ethIP_arphdr* arp, unsigned char* senderMAC, 
                 unsigned char* senderIP, unsigned char* targetIP) {
    memset(arp, 0, sizeof(struct ethIP_arphdr));
arp->arhdr.ar_hrd = htons(ARPHRD_ETHER);
arp->arhdr.ar_pro = htons(ETH_P_IP);
ar->arhdr.ar_hln = 6;
ar->arhdr.ar_pln = 4;
memcpy(arp->ar_sha,senderMAC,6);
memcpy(arp->ar_sip,senderIP,4);
memcpy(arp->ar_tip,targetIP,4);
}

unsigned char* getIPHeader(struct iphdr* ip,unsigned char* buf) {
    memcpy(ip,buf,sizeof(struct iphdr));
    return(buf+sizeof(struct iphdr));
}

unsigned char* getUDPHeader(struct udphdr* udp,unsigned char* buf) {
    memcpy(udp,buf,sizeof(struct udphdr));
    udp->source = ntohs(udp->source);
    udp->dest = ntohs(udp->dest);
    return(buf+sizeof(struct udphdr));
}

unsigned char* getTCPHeader(struct tcphdr* tcp,unsigned char* buf) {
    memcpy(tcp,buf,sizeof(struct tcphdr));
    tcp->source = ntohs(tcp->source);
    tcp->dest = ntohs(tcp->dest);
    return(buf+sizeof(struct tcphdr));
}

void getARPHeader(struct ethIP_arphdr* arp,unsigned char* buf) {
    memcpy(arp,buf,sizeof(struct ethIP_arphdr));
}
APPENDIX B
Alpha Shield Specification

ASSUMPTIONS

- The PC port is connected to a 10Mbps Ethernet device.
- The cable/DSL port is connected to a 10Mbps Ethernet device.

ENVIRONMENT VARIABLES

inboundLED: OFF,GREEN,FLASHING_GREEN,RED,FLASHING_RED
outboundLED: OFF,GREEN,FLASHING_GREEN,RED,FLASHING_RED
connectLED: GREEN,FLASHING_GREEN,RED
modeSwitch: MANUAL,AUTO,LOCK

STATE VARIABLES

sessionTable: set of (id: sessionId, timer: int)

TYPES

sessionId: (protocol: TCP,UDP,IP,srcIP,srcPort,dstIP,dstPort)

EVENTS

connect: blue button pressed
disconnect: green button pressed
startup: the Alpha Shield is powered on

TRANSITIONS AND OUTPUTS

startup:
  sessionTable = {}

outbound frame F:
  if F is an ARP frame
    accept F
  else if F.dstMAC is broadcast
    reject F
  else if F is an IP frame
    accept F
    if getId(F) in sessionTable
      set timer to 0
    else
add (getId(F),0) to sessionTable
else
    reject F?

inboundFrame F:
    if F is an ARP frame
        accept F
    else if F.dstMAC is broadcast
        reject F
    else if F is a TCP frame and F.srcPort = 20 and
    (exists P) (TCP,F.dstIP,P,F.srcIP,21) in sessionTable
        accept F
    else if F is a TCP, UDP or IP frame
        if swapId(getId(F)) in sessionTable
            accept F
        else
            reject?
    clock tick (at 1 Hz.):
        for each entry s in sessionTable
            if s.timer == 300
                remove s from sessionTable
            else
                s.timer++

FUNCTIONS

sessionId getId(f)
    if f is TCP
        return (TCP,f.srcIP,f.srcPort,f.dstIP,f.dstPort)
    else if f is UDP
        return (UDP,f.srcIP,f.srcPort,f.dstIP,f.dstPort)
    else if f is IP
        return (TCP,f.srcIP,f.0,f.dstIP,0)
sessionId swap(id:sessionId)
    return (id.protocol,id.dstIP,id.dstPort,id.srcIP,id.srcPort)

DELAY
TCP, UDP and IP outbound or inbound frames:
\[ \text{delay} = 1.1 \times \text{frameLength} - \text{overhead} \text{ (overhead = 59 to 67 \(\mu\text{sec}\))} \]

ARP outbound or inbound frames:
\[ \text{delay} = 114 \ \mu\text{sec} \]

CAPACITY

The session table can hold at most 96 entries.
APPENDIX C

Alpha Shield Test Suite code

C.0.5 AlphaShield.c

```c
#include<stdio.h>
#include<string.h>
#include<netinet/ip.h>
#include<netinet/udp.h>
#include<netinet/tcp.h>
#include<net/if_arp.h>
#include<linux/if_packet.h>
#include<linux/if_ether.h>
#include "socketLibrary.h"
#include "headerUtility.h"

#define FRAMELENGTH 64

int fdIn,fdoOut;

//constants for addresses and ports
const unsigned char MAC1[6] = {0x00,0x00,0x00,0x00,0x00,0x01};
const unsigned char MAC2[6] = {0x00,0x00,0x00,0x00,0x00,0x02};
const unsigned char BCASTMAC[6] = {0xff,0xff,0xff,0xff,0xff,0xff};

const unsigned char IP1[4] = {10,1,1,1};
const unsigned char IP2[4] = {10,2,1,2};
const unsigned char IP3[4] = {10,2,1,3};

const unsigned short PORT1 = 2000;
const unsigned short PORT2 = 2001;

//send an ethernet frame on fd
//preconditions
```
void sendFrame(int fd, void* frame, int len)
{
    ss_send(fd, (unsigned char*)frame, len);
}

int ARPCompare(void* expectedFrame, void* actualFrame)
{
    struct ethIP_arphdr* actualARPHdr;
    struct ethIP_arphdr* expectedARPHdr;

    getARPHeader(expectedARPHdr, expectedFrame);
    getARPHeader(actualARPHdr, actualFrame);

    if (actualARPHdr->arphdr.ar_op == expectedARPHdr->arphdr.ar_op)
        return(1);
    else
        return(0);
}

int UDPCompare(void* expectedFrame, void* actualFrame)
{
    struct iphdr actualIPHdr;
    struct iphdr expectedIPHdr;
    struct udphdr actualUDPHdr;
    struct udphdr expectedUDPHdr;

    getUDPHeader(&expectedUDPHdr, getIPHeader(&expectedIPHdr, expectedFrame + 14));
    getUDPHeader(&actualUDPHdr, getIPHeader(&actualIPHdr, actualFrame));

    if ((actualIPHdr.saddr == expectedIPHdr.saddr) &&
        (actualIPHdr.daddr == expectedIPHdr.daddr) &&
        (actualIPHdr.protocol == expectedIPHdr.protocol) &&

(expectedUDPHdr.source == actualUDPHdr.source) &&
(expectedUDPHdr.dest == actualUDPHdr.dest))
    return(1);
else
    return(0);
}

//compare 2 ip frames and return 1 if they have the same session Id and 0 if different
int IPCompare(void* expectedFrame,void* actualFrame)
{
    struct iphdr actualIPHdr;
    struct iphdr expectedIPHdr;

    getIPHeader(&expectedIPHdr,expectedFrame+14);
    getIPHeader(&actualIPHdr,actualFrame);

    if ((actualIPHdr.saddr == expectedIPHdr.saddr) &&
      (actualIPHdr.daddr == expectedIPHdr.daddr) &&
      (actualIPHdr.protocol == expectedIPHdr.protocol))
        return(1);
    else
        return(0);
}

//compare 2 tcp frames and return 1 if they have the same session ID and 0 if different
int TCPCompare(void* expectedFrame,void* actualFrame)
{
    struct iphdr actualIPHdr;
    struct iphdr expectedIPHdr;
    struct tcphdr actualTCPHdr;
    struct tcphdr expectedTCPHdr;

    getTCPHeader(&expectedTCPHdr,getIPHeader(&expectedIPHdr,expectedFrame+14));
    getTCPHeader(&actualTCPHdr,getIPHeader(&actualIPHdr,actualFrame));
if (((actualIPHdr.saddr == expectedIPHdr.saddr) &&
    (actualIPHdr.daddr == expectedIPHdr.daddr) &&
    (actualIPHdr.protoco1 == expectedIPHdr.protocol) &&
    (expectedTCPHdr.source == actualTCPHdr.source) &&
    (expectedTCPHdr.dest == actualTCPHdr.dest))
    return(1);
else
    return(0);
}

//expect a frame on fd and return 1 if expected frame arrives or 0 if
//expected frame does not arrive. No frame is expected when compare = NULL
int Expect(int fd,void* expectedFrame,int (*compare)(void*,void*))
{
    struct ethhdr ethhdr;
    unsigned char buffer[1514];

    if (compare == NULL) {
        if (ss_receive(fd,&ethhdr,buffer,5) > 0)
            return(0);
        else
            return(1);
    } else {
        if (ss_receive(fd,&ethhdr,buffer,5) <= 0)
            return(0);
        else {
            if((*compare)(expectedFrame,buffer))
                return(1);
            else
                return(0);
        }
    }
}

//initializes an ARP frame
void initARPFrame(unsigned char* frame) {
    struct ss_ethhdr ether = (struct ethhdr*)(frame);
    struct ethIP_arphdr arp = (struct ethIP_arphdr*)(frame+14);

    ss_setEthernetHeader(&ether,MAC2,MAC1,ETH_P_ARP);
    setARPHdr(&arp,MAC2,IP2,IP1);

    memcpy(frame,&ether,sizeof(struct ethhdr));
    memcpy(frame+sizeof(struct ethhdr),&arp,sizeof(struct ethIP_arphdr));
}

void initInUDP(unsigned char* frame) {
    struct ethhdr ethhdr;
    struct iphdr iphdr;
    struct udphdr udphdr;

    ss_setEthernetHeader(&ethhdr,MAC1,MAC2,ETH_P_IP);
    setIPHeader(&iphdr,IP1,IP2,IPPROTO_UDP,FRAMELENGTH);
    setUDPHdr(&udphdr,PORT1,PORT2,FRAMELENGTH);

    memcpy(frame,&ethhdr,sizeof(ethhdr));
    memcpy(frame + sizeof(ethhdr),&iphdr,sizeof(iphdr));
    memcpy(frame + sizeof(ethhdr) + sizeof(iphdr),&udphdr,sizeof(udphdr));
}

void initOutUDP(unsigned char* frame)
{  
    struct ethhdr ethhdr;
    struct iphdr iphdr;
    struct udphdr udphdr;

    ss_setEthernetHeader(&ethhdr, MAC2, MAC1, ETH_P_IP);
    setIPHeader(&iphdr, IP2, IP1, IPPROTO_UDP, FRAMELENGTH);
    setUDPHdr(&udphdr, PORT2, PORT1, FRAMELENGTH);

    memcpy(frame, &ethhdr, sizeof(ethhdr));
    memcpy(frame + sizeof(ethhdr), &iphdr, sizeof(iphdr));
    memcpy(frame + sizeof(ethhdr) + sizeof(iphdr), &udphdr, sizeof(udphdr));
}

// initializes an inbound IP frame
// preconditions
// *frame is an allocated buffer of at least 64 bytes
void initInIP(unsigned char* frame)
{
    struct ethhdr ethhdr;
    struct iphdr iphdr;

    ss_setEthernetHeader(&ethhdr, MAC1, MAC2, ETH_P_IP);
    setIPHeader(&iphdr, IP1, IP2, IPPROTO_IP, FRAMELENGTH);

    memcpy(frame, &ethhdr, sizeof(ethhdr));
    memcpy(frame + sizeof(ethhdr), &iphdr, sizeof(iphdr));
}

// initializes an outbound IP frame
// preconditions
// *frame is an allocated buffer of at least 64 bytes
void initOutIP(unsigned char* frame)
{
    struct ethhdr ethhdr;
}
struct iphdr iphdr;

ss_setEthernetHeader(&ethhdr, MAC2, MAC1, ETH_P_IP);
setIPHeader(&iphdr, IP2, IP1, IPPROTO_TCP, FRAMELENGTH);

memcpy(frame, &ethhdr, sizeof(ethhdr));
memcpy(frame + sizeof(ethhdr), &iphdr, sizeof(iphdr));
}

// initializes an inbound TCP frame
// preconditions
// *frame is an allocated buffer of at least 64 bytes
void initInTCP(unsigned char* frame)
{
    struct ethhdr ethhdr;
    struct iphdr iphdr;
    struct tcphdr tcphdr;

    ss_setEthernetHeader(&ethhdr, MAC1, MAC2, ETH_P_IP);
    setIPHeader(&iphdr, IP1, IP2, IPPROTO_TCP, FRAMELENGTH);
    setTCPHeader(&tcphdr, PORT1, PORT2);

    memcpy(frame, &ethhdr, sizeof(ethhdr));
    memcpy(frame + sizeof(ethhdr), &iphdr, sizeof(iphdr));
    memcpy(frame + sizeof(ethhdr) + sizeof(iphdr), &tcphdr, sizeof(tcphdr));
}

// initializes an outbound TCP frame
// preconditions
// *frame is an allocated buffer of at least 64 bytes
void initOutTCP(unsigned char* frame)
{
    struct ethhdr ethhdr;
    struct iphdr iphdr;
    struct tcphdr tcphdr;

ss_setEthernetHeader(&ethhdr, MAC2, MAC1, ETH_P_IP);
ssetUpHeader(&iphdr, IP2, IP1, IPPROTO_TCP, FRAMELENGTH);
setTCPHeader(&tcphdr, PORT2, PORT1);

memcpy(frame, &ethhdr, sizeof(ethhdr));
memcpy(frame + sizeof(ethhdr), &iphdr, sizeof(iphdr));
memcpy(frame + sizeof(ethhdr) + sizeof(iphdr), &tcphdr, sizeof(tcphdr));

// test outbound arp requests and replies
void outboundARP()
{
    unsigned char frame[60];
    struct ethhdr* eth = (struct ethhdr*)(frame);
    struct ethIP_arphdr* arp = (struct ethIP_arphdr*)(frame+14);

    initARPFrame(frame);

    // *** ARP request
    arp->arphdr.ar_op = htons(ARPOP_REQUEST);

    // broadcast dest MAC
    memcpy(eth->h_dest, BCASTMAC, 6);
    sendFrame(fdOut, frame, 60);
    Expect(fdIn, frame, &ARPCompare);

    // non broadcast dest MAC
    memcpy(eth->h_dest, MAC2, 6);
    sendFrame(fdOut, frame, 60);
    Expect(fdIn, frame, &ARPCompare);

    // *** ARP reply
    arp->arphdr.ar_op = htons(ARPOP_REPLY);
    memcpy(arp->ar_tha, MAC1, 6);
//broadcast dest MAC
memcpy(eth->h_dest,BCASTMAC,6);
sendFrame(fdOut,frame,60);
Expect(fdIn,frame,&ARPCompare);

//non broadcast dest MAC
memcpy(eth->h_dest,MAC2,6);
sendFrame(fdOut,frame,60);
Expect(fdIn,frame,&ARPCompare);
}

//test inbound arp requests and replies
void inboundARP()
{
  unsigned char frame[60];
  struct ethhdr* eth = (struct ethhdr*)(frame);
  struct ethIP_arphdr* arp = (struct ethIP_arphdr*)(frame+14);

  initARPFrame(frame);

  //*** ARP request
  arp->arphdr.ar_op = htons(ARP0P_REQUEST);

  //broadcast dest MAC
  memcpy(eth->h_dest,BCASTMAC,6);
  sendFrame(fdIn,frame,60);
  Expect(fdOut,frame,&ARPCompare);

  //non broadcast dest MAC
  memcpy(eth->h_dest,MAC2,6);
  sendFrame(fdIn,frame,60);
  Expect(fdOut,frame,&ARPCompare);

  //*** ARP reply
arp->arphdr.ar_op = htons(ARPOP_REPLY);
memcpy(arp->ar_tha,MAC2,6);

// broadcast dest MAC
memcpy(eth->h_dest,BCASTMAC,6);
sendFrame(fdIn,frame,60);
Expect(fdOut,frame,&ARPCompare);

// non broadcast dest MAC
memcpy(eth->h_dest,MAC2,6);
sendFrame(fdIn,frame,60);
Expect(fdOut,frame,&ARPCompare);

} // tests in and outbound UDP frames

void UDP() {

    // define and initialise frame buffers and header pointers
    unsigned char outFrame[1514];
    struct ethhdr* outEth = (struct ethhdr*) outFrame;
    struct iphdr* outIP = (struct iphdr*) (outFrame + sizeof(struct ethhdr));
    struct udphdr* outUDP = (struct udphdr*) (outFrame + sizeof(struct ethhdr) +
                                 sizeof(struct iphdr));

    unsigned char inFrame[1514];
    struct ethhdr* inEth = (struct ethhdr*) inFrame;
    struct iphdr* inIP = (struct iphdr*) (inFrame + sizeof(struct ethhdr));
    struct udphdr* inUDP = (struct udphdr*) (inFrame + sizeof(struct ethhdr) +
                                 sizeof(struct iphdr));

    initOutUDP(outFrame);
    initInUDP(inFrame);

    // outbound UDP frame
    sendFrame(fdOut,outFrame,FRAMELENGTH);
Expect(fdIn,outFrame,&UDPCmp);

// inbound UDP frame with swapped session id
sendFrame(fdIn,inFrame,FRAMELENGTH);
Expect(fdOut,inFrame,&UDPCmp);

// inbound UDP frame with not swapped session id
memcpy(&(inEth->saddr),IP3,4);
setIPChecksum(inIP);
sendFrame(fdIn,inFrame,FRAMELENGTH);
Expect(fdOut,inFrame,NULL);

// outbound UDP with broadcast dest MAC
memcpy(outEth->h_dest,BCASTMAC,6);
sendFrame(fdOut,outFrame,FRAMELENGTH);
Expect(fdIn,outFrame,NULL);

// inbound UDP with broadcast dest MAC
initInUDP(inFrame);
memcpy(inEth->h_dest,BCASTMAC,6);
sendFrame(fdIn,inFrame,FRAMELENGTH);
Expect(fdOut,inFrame,NULL);

// inbound TCP frame with swapped session id
initInTCP(inFrame);
sendFrame(fdIn,inFrame,FRAMELENGTH);
Expect(fdOut,inFrame,NULL);

// inbound IP frame with swapped IP addresses
initInIP(inFrame);
sendFrame(fdIn,inFrame,FRAMELENGTH);
Expect(fdOut,inFrame,NULL);
}

// tests in and outbound IP frames
void IP()
{
    //define and initialise frame buffers and header pointers
    unsigned char outFrame[1514];
    struct ethhdr* outEth = (struct ethhdr*)outFrame;
    struct iphdr* outIP = (struct iphdr*)(outFrame + sizeof(struct ethhdr));

    unsigned char inFrame[1514];
    struct ethhdr* inEth = (struct ethhdr*)inFrame;
    struct iphdr* inIP = (struct iphdr*)(inFrame + sizeof(struct ethhdr));

    initOutIP(outFrame);
    iniInIP(inFrame);

    //outbound IP frame
    sendFrame(fdOut,outFrame,FRAMELENGTH);
    Expect(fdIn,outFrame,&IPCompare);

    //inbound IP frame with swapped session id
    sendFrame(fdIn,inFrame,FRAMELENGTH);
    Expect(fdOut,inFrame,&IPCompare);

    //inbound IP frame with not swapped session id
    memcpy(&(inIP->saddr),IP3,4);
    setIPCKsum(inIP);
    sendFrame(fdIn,inFrame,FRAMELENGTH);
    Expect(fdOut,inFrame,NULL);

    //outbound IP with broadcast dest MAC
    memcpy(outEth->h_dest,BCASTMAC,6);
    sendFrame(fdOut,outFrame,FRAMELENGTH);
    Expect(fdIn,outFrame,NULL);

    //inbound IP with broadcast dest MAC
    initInIP(inFrame);
\texttt{memcpy}(\texttt{inEth->h_dest}, \texttt{BCASTMAC}, 6);
\texttt{sendFrame(fdIn, inFrame, FRAMELENGTH);}
\texttt{Expect(fdOut, inFrame, NULL);}

// inbound TCP frame with swapped session id
\texttt{initInTCP(inFrame);}
\texttt{sendFrame(fdIn, inFrame, FRAMELENGTH);}
\texttt{Expect(fdOut, inFrame, NULL);}

// inbound UDP frame with swapped session id
\texttt{initInUDP(inFrame);}
\texttt{sendFrame(fdIn, inFrame, FRAMELENGTH);}
\texttt{Expect(fdOut, inFrame, NULL);}

// tests in and outbound TCP frames
\texttt{void TCP()}
{

// define and initialise frame buffers and header pointers
\texttt{unsigned char outFrame[1514];}
\texttt{struct ethhdr* outEth = (struct ethhdr*)outFrame;}
\texttt{struct iphdr* outIP = (struct iphdr*)(outFrame + sizeof(struct ethhdr));}
\texttt{struct tcphdr* outTCP = (struct tcphdr*)(outFrame + sizeof(struct ethhdr) +
sizeof(struct iphdr));}

\texttt{unsigned char inFrame[1514];}
\texttt{struct ethhdr* inEth = (struct ethhdr*)inFrame;}
\texttt{struct iphdr* inIP = (struct iphdr*)(inFrame + sizeof(struct ethhdr));}
\texttt{struct tcphdr* inTCP = (struct tcphdr*)(inFrame + sizeof(struct ethhdr) +
sizeof(struct iphdr));}

\texttt{initOutTCP(outFrame);}
\texttt{initInTCP(inFrame);}

// outbound TCP frame
sendFrame(fdOut, outFrame, FRAMELENGTH);
Expect(fdIn, outFrame, &TCPCompare);

//inbound TCP frame with swapped session id
sendFrame(fdIn, inFrame, FRAMELENGTH);
Expect(fdOut, inFrame, &TCPCompare);

//inbound TCP frame with not swapped session id
memcpy(&inIP->saddr), IP3, 4);
setIPChecksum(inIP);
sendFrame(fdIn, inFrame, FRAMELENGTH);
Expect(fdOut, inFrame, NULL);

//outbound TCP with broadcast dest MAC
memcpy(outEth->h_dest, BCASTMAC, 6);
sendFrame(fdOut, outFrame, FRAMELENGTH);
Expect(fdIn, outFrame, NULL);

//inbound TCP with broadcast dest MAC
initInTCP(inFrame);
memcpy(inEth->h_dest, BCASTMAC, 6);
sendFrame(fdIn, inFrame, FRAMELENGTH);
Expect(fdOut, inFrame, NULL);

//inbound UDP frame with swapped session id
initInUDP(inFrame);
sendFrame(fdIn, inFrame, FRAMELENGTH);
Expect(fdOut, inFrame, NULL);

//inbound IP frame with swapped IP addresses
initInIP(inFrame);
sendFrame(fdIn, inFrame, FRAMELENGTH);
Expect(fdOut, inFrame, NULL);
void FTP()
{
//define and initialise frame buffers and header pointers
unsigned char outFrame[1514];
struct ethhdr* outEth = (struct ethhdr*)outFrame;
struct iphdr* outIP = (struct iphdr*)(outFrame + sizeof(struct ethhdr));
struct tcphdr* outTCP = (struct tcphdr*)(outFrame + sizeof(struct ethhdr) +
    sizeof(struct iphdr));

unsigned char inFrame[1514];
struct ethhdr* inEth = (struct ethhdr*)inFrame;
struct iphdr* inIP = (struct iphdr*)(inFrame + sizeof(struct ethhdr));
struct tcphdr* inTCP = (struct tcphdr*)(inFrame + sizeof(struct ethhdr) +
    sizeof(struct iphdr));

initOutTCP(outFrame);
initInTCP(inFrame);

//outbound FTP control frame
outTCP->dest = htons(21);
sendFrame(fdOut,outFrame,FRAMELENGTH);
Expect(fdIn,outFrame,&TCPCompare);

//inbound FTP data frame
inTCP->source = htons(20);
inTCP->dest = htons(2002);
sendFrame(fdIn,inFrame,FRAMELENGTH);
Expect(fdOut,inFrame,&TCPCompare);
}

//verify that the session table entry times out after 5 minutes
sessionTableTimeout()
{
unsigned char inFrame[1514];
struct udphdr* inUDP = (struct udphdr*)(inFrame + sizeof(struct ethhdr) +
sizeof(struct iphdr));

unsigned char outFrame[1541];
struct udphdr* outUDP = (struct udphdr*)(outFrame + sizeof(struct ethhdr) +
    sizeof(struct iphdr));

initOutUDP(outFrame);
initInUDP(inFrame);

//send outbound UDP frame
sendFrame(fdOut,outFrame,FRAMELENGTH);
Expect(fdIn,outFrame,&UDPCompare);

//send inbound UDP frame
sendFrame(fdIn,inFrame,FRAMELENGTH);
Expect(fdOut,inFrame,&UDPCompare);

//sleep for 5 minutes
sleep(300);

//send inbound UDP frame, should be dropped
sendFrame(fdIn,inFrame,FRAMELENGTH);
Expect(fdOut,inFrame,NULL);
}

//verify that the size of the session table is 96
sessionTableSize()
{
    unsigned char inFrame[1514];
    struct udphdr* inUDP = (struct udphdr*)(inFrame + sizeof(struct ethhdr) +
        sizeof(struct iphdr));

    unsigned char outFrame[1541];
    struct udphdr* outUDP = (struct udphdr*)(outFrame + sizeof(struct ethhdr) +
        sizeof(struct iphdr));
int i, count=0;

initOutUDP(outFrame);
initInUDP(inFrame);

//send out 97 outbound frames with different source ports
for(i=0; i<97; i++) {
    sendFrame(fdOut, outFrame, FRAMELENGTH);
    Expect(fdIn, outFrame, &UDPCompare);
    outUDP->source = htons(ntohs(outUDP->source) + 1);
}

//send out 97 inbound UDP frame with swapped session ids
for(i=0; i<97; i++) {
    sendFrame(fdIn, inFrame, FRAMELENGTH);
    if(Expect(fdOut, inFrame, &UDPCompare))
        count += 1;
    inUDP->dest = htons(ntohs(inUDP->dest) + 1);
}
printf("The size of the session table is %d\n", count);

main(int argc, char* argv[])
{
    //create and bind inbound and outbound sockets
    fdIn = ss_socket();
    ss_bind(fdIn,"eth2");

    fdOut = ss_socket();
    ss_bind(fdOut,"eth1");

    outboundARP();

    inboundARP();
UDP();

// Need to reset alpha shield between these 2 tests
IP();

// Need to reset alpha shield between these 2 tests
TCP();

FTP();

sessionTableTimeout();

sessionTableSize();

// close fdIn and fdOut
ss_close(fdIn);
ss_close(fdOut);
}

C.0.6  DOS.c
#include<stdio.h>
#include<string.h>
#include<linux/if_packet.h>
#include<linux/if_ether.h>
#include<net/if_arp.h>
#include "socketLibrary.h"
#include "headerUtility.h"

unsigned char MAC1[6] = {0x00, 0x50, 0xBF, 0x91, 0x00, 0x00};
unsigned char MAC2[6] = {0x00, 0x60, 0x97, 0xCC, 0x27, 0x22};
unsigned char IP1[4] = {10, 1, 0, 0};
unsigned char IP2[4] = {10, 0, 0, 1};

// increment MAC/IP address by 1
void incrementAddress(unsigned char* byte1, unsigned char* byte2)
{
    if (*byte2 < 255)
        *byte2 += 1;
    else{
        if (*byte1 < 255){
            *byte2 = 0;
            *byte1 += 1;
        }
    }
}

void initARPReply(unsigned char* frame){
    struct ethhdr ether = (struct ethhdr*) (frame);
    struct ethIP_arphdr arp = (struct ethIP_arphdr*) (frame+14);
    ss_setEthernetHeader(&ether, MAC2, MAC1, ETH_P_ARP);
    setARPHeader(&arp, MAC2, IP2, IP1);
    arp->arp->arp_op = htons(ARP0P_REPLY);
    memcpy(arp->arp->tha, MAC1, 6);
    memcpy(frame, &ether, sizeof(struct ethhdr));
    memcpy(frame+sizeof(struct ethhdr), &arp, sizeof(struct ethIP_arphdr));
}

main(int argc, char* argv[])
{
    int fd;
    unsigned char frame[1514];

    // create socket and bind to eth1
    fd = ss_socket();
    ss_bind(fd, "eth1");
    incrementAddress(&sourceIP[2],&sourceIP[3]);
    incrementAddress(&sourceMACAddr[4],&sourceMACAddr[5]);
    initARPReply(frame);
    ss_send(fd,(unsigned char*)frame,60);
    usleep(100);
}
ss_close(fd);
APPENDIX D
iptables Test Suite code

D.0.7 iptables.c

#include<stdio.h>
#include "socketLibrary.h"
#include "headerUtility.h"

#define FRAMELENGTH 70

int fd1, fd2;

// constants for addresses and ports
const unsigned char MACl[6] = {0x00,0x00,0x00,0x00,0x00,0x01};
const unsigned char MAC2[6] = {0x00,0x50,0xBF,0x91,0xC0,0xA5};
const unsigned char MAC3[6] = {0x00,0x50,0xBF,0x91,0xAD,0x9F};
const unsigned char MAC4[6] = {0x00,0x00,0x00,0x00,0x00,0x02};
const unsigned char IP1[4] = {10,1,0,1};
const unsigned char IP2[4] = {10,2,0,2};
const unsigned char IP3[4] = {10,2,0,3};
const unsigned char IP4[4] = {10,1,0,2};
const unsigned char IP5[4] = {10,2,0,3};
const unsigned char IP6[4] = {10,2,0,4};
const unsigned char IP7[4] = {10,2,0,5};
const unsigned char IP8[4] = {10,1,0,3};
const unsigned char IP9[4] = {10,1,0,4};
const unsigned char IP10[4] = {10,1,0,5};
const unsigned char IP11[4] = {10,2,2,2};
const unsigned short PORT1 = 2000;
const unsigned short PORT2 = 2001;
//send an ethernet frame on fd
//preconditions
// frame points to at least len bytes
void sendFrame(int fd, void* frame, int len)
{
    ss_send(fd, (unsigned char*)frame, len);
}

//compare 2 udp frames and return 1 if they are identical and 0 if different
int UDPCompare(void* expectedFrame, void* actualFrame)
{
    struct iphdr actualIPHdr;
    struct iphdr expectedIPHdr;
    struct udphdr actualUDPHdr;
    struct udphdr expectedUDPHdr;

    getUDPHeader(&expectedUDPHdr, getIPHeader(&expectedIPHdr, expectedFrame+14));
    getUDPHeader(&actualUDPHdr, getIPHeader(&actualIPHdr, actualFrame));

    if ((actualIPHdr.saddr == expectedIPHdr.saddr) &&
        (actualIPHdr.daddr == expectedIPHdr.daddr) &&
        (actualIPHdr.protocol == expectedIPHdr.protocol) &&
        (expectedUDPHdr.source == actualUDPHdr.source) &&
        (expectedUDPHdr.dest == actualUDPHdr.dest))
        return(1);
    else
        return(0);
}

//compare 2 tcp frames and return 1 if they are identical and 0 if different
int TCPCompare(void* expectedFrame, void* actualFrame)
{
    struct iphdr actualIPHdr;
    struct iphdr expectedIPHdr;
    struct tcphdr actualTCPHdr;

struct tcphdr expectedTCPHdr;

getTCPHeader(&expectedTCPHdr, getIPHeader(&expectedIPHdr, expectedFrame+14));
getTCPHeader(&actualTCPHdr, getIPHeader(&actualIPHdr, actualFrame));

if ((actualIPHdr.saddr == expectedIPHdr.saddr) &&
    (actualIPHdr.daddr == expectedIPHdr.daddr) &&
    (actualIPHdr.protocol == expectedIPHdr.protocol) &&
    (expectedTCPHdr.source == actualTCPHdr.source) &&
    (expectedTCPHdr.dest == actualTCPHdr.dest))
    return(1);
else
    return(0);
}

//expect a frame on fd and return 1 if expected frame arrives or 0 if
//expected frame does not arrive. No frame is expected when compare = NULL

int Expect(int fd, void* expectedFrame, int (*compare)(void*, void*))
{
    struct ethhdr ethhdr;
    unsigned char buffer[1514];

    if (compare == NULL) {
        if (ss_receive(fd, &ethhdr, buffer, 5) > 0)
            return(0);
        else
            return(1);
    }
    else {
        if (ss_receive(fd, &ethhdr, buffer, 5) <= 0)
            return(0);
        else {
            if (((*compare)(expectedFrame, buffer))
                return(1);
            else
//initializes a UDP frame to be sent out on eth1
//preconditions
// *frame is an allocated buffer of at least 64 bytes
void initUDPEth1(unsigned char* frame)
{
    struct ethhdr ethhdr;
    struct iphdr iphdr;
    struct udphdr udphdr;

    ss_setEthernetHeader(&ethhdr, MAC2, MAC1, ETH_P_IP);
    setIPHeader(&iphdr, IP2, IP1, IPPROTO_UDP, FRAMELENGTH);
    setUDPHdr(&udphdr, PORT2, PORT1, FRAMELENGTH);

    memcpy(frame, &ethhdr, sizeof(ethhdr));
    memcpy(frame + sizeof(ethhdr), &iphdr, sizeof(iphdr));
    memcpy(frame + sizeof(ethhdr) + sizeof(iphdr), &udphdr, sizeof(udphdr));
}

//initializes a UDP frame to be sent out on eth2
//preconditions
//  *frame is an allocated buffer of at least 64 bytes
void initUDPEth2(unsigned char* frame)
{
    struct ethhdr ethhdr;
    struct iphdr iphdr;
    struct udphdr udphdr;

    ss_setEthernetHeader(&ethhdr, MAC3, MAC4, ETH_P_IP);
    setIPHeader(&iphdr, IP2, IP1, IPPROTO_UDP, FRAMELENGTH);
    setUDPHdr(&udphdr, PORT2, PORT1, FRAMELENGTH);

    memcpy(frame, &ethhdr, sizeof(ethhdr));
    memcpy(frame + sizeof(ethhdr), &iphdr, sizeof(iphdr));
    memcpy(frame + sizeof(ethhdr) + sizeof(iphdr), &udphdr, sizeof(udphdr));

    return(0);
}
memcpy(frame,&ethhdr,sizeof(ethhdr));
memcpy(frame + sizeof(ethhdr),&iphdr,sizeof(iphdr));
memcpy(frame + sizeof(ethhdr) + sizeof(iphdr),&tcpdr,sizeof(tcpdr));
}

//initializes a TCP frame to be sent out on eth1
//preconditions
// *frame is an allocated buffer of at least 64 bytes
void initTCPeth1(unsigned char* frame)
{
    struct ethhdr ethhdr;
    struct iphdr iphdr;
    struct tcphdr tcpdr;

    ss_setEthernetHeader(&ethhdr,MAC3,MAC4,ETH_P_IP);
    setIPHeader(&iphdr,IP1,IP2,IPPROTO_TCP,FRAMELENGTH);
    setTCPHeader(&tcpdr,PORT2,PORT1);

    memcpy(frame,&ethhdr,sizeof(ethhdr));
    memcpy(frame + sizeof(ethhdr),&iphdr,sizeof(iphdr));
    memcpy(frame + sizeof(ethhdr) + sizeof(iphdr),&tcpdr,sizeof(tcpdr));
}

//initializes a TCP frame to be sent out on eth2
//preconditions
// *frame is an allocated buffer of at least 64 bytes
void initTCPeth2(unsigned char* frame)
{
    struct ethhdr ethhdr;
    struct iphdr iphdr;
    struct tcphdr tcpdr;

    ss_setEthernetHeader(&ethhdr,MAC3,MAC4,ETH_P_IP);
    setIPHeader(&iphdr,IP1,IP2,IPPROTO_TCP,FRAMELENGTH);
setTCPHeader(&tcphdr, PORT1, PORT2);

memcpy(frame, &ethhdr, sizeof(ethhdr));
memcpy(frame + sizeof(ethhdr), &iphdr, sizeof(iphdr));
memcpy(frame + sizeof(ethhdr) + sizeof(iphdr), &tcphdr, sizeof(tcphdr));
}

//flush all iptables rules and set the FORWARD policy to drop
void initIPTables()
{
    //flush all rules
    system("ssh -1 142.104.101.147 iptables -P");

    //set FORWARD policy to DROP
    system("ssh -1 142.104.101.147 iptables --policy FORWARD DROP");
}

//flush all nat rules and set the FORWARD policy to accept
void initNATTable()
{
    //flush all rules
    system("ssh -l 142.104.101.147 iptables -t nat -F");

    //set FORWARD policy to ACCEPT
    system("ssh -l 142.104.101.147 iptables --policy FORWARD ACCEPT");
}

//tests the -i match
inInterfaceTest()
{
    unsigned char frame[1514];

    initIPTables();

    //set up rule to accept frames entering on eth1

system("ssh -l 142.104.101.147 iptables -A FORWARD -i eth1 -j ACCEPT");

//create outbound UDP frame
initUDPEth1(frame);

//send out udp frame on eth1 -- should get thru
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&UDPCompare);

initUDPEth2(frame);

//send out udp frame on eth2 -- should not get thru
sendFrame(fd2,frame,FRAMELENGTH);
Expect(fd1,frame,NULL);
}

//test the -o match
outInterfaceTest()
{
    unsigned char frame[1514];

    initIPTables();

    //set up rule to accept all frames having out-interface eth1
    system("ssh -l 142.104.101.147 iptables -A FORWARD -o eth1 -j ACCEPT");

    initUDPEth1(frame);

    //send out udp frame on eth1 -- should be dropped
    sendFrame(fd1,frame,FRAMELENGTH);
    Expect(fd2,frame,NULL);

    initUDPEth2(frame);

    //send out udp frame on eth2 -- should not get thru
sendFrame(fd2, frame, FRAMELENGTH);
Expect(fd1, frame, &UDPCompare);
}

// tests the -p match
protocolTest()
{
  unsigned char tcpFrame[1514];
  unsigned char udpFrame[1514];

  initUDPEth1(udpFrame);
  initTCPEth1(tcpFrame);

  initIPTables();

  // set rule to accept only udp frames
  system("ssh -l 142.104.101.147 iptables -A FORWARD -p udp -j ACCEPT");

  // send udp frame on eth1 -- should get thru
  sendFrame(fd1, udpFrame, FRAMELENGTH);
  Expect(fd2, udpFrame, &UDPCompare);

  // send tcp frame on eth1 -- should not get thru
  sendFrame(fd1, tcpFrame, FRAMELENGTH);
  Expect(fd2, tcpFrame, NULL);

  // set rule to accept frames of all protocols
  system("ssh -l 142.104.101.147 iptables -A FORWARD -p all -j ACCEPT");

  // send tcp frame on eth1 -- should get thru this time
  sendFrame(fd1, tcpFrame, FRAMELENGTH);
  Expect(fd2, tcpFrame, &TCPCompare);
}

// tests the -s match
sourceAddressTest()
{
    unsigned char udpFrame[1514];
    struct iphdr* ip = (struct iphdr*)(udpFrame + sizeof(struct ethhdr));
    initUDPEth1(udpFrame);

    //initIPTables();

    //set rule to accept frames having source ip address 10.1.0.1
    system("ssh -1 142.104.101.147 iptables -A FORWARD -s 10.1.0.1 -j ACCEPT");

    //send udp frame with source ip address 10.1.0.1 -- should get thru
    sendFrame(fd1,udpFrame,FRAMELENGTH);
    Expect(fd2,udpFrame,&UDPCompare);

    //change source ip address of udpFrame to 10.1.0.2
    memcpy(&(ip->saddr),IP4,4);
    setIPCKsum(ip);

    //send udp frame with source ip address 10.1.0.2 -- should not get thru
    sendFrame(fd1,udpFrame,FRAMELENGTH);
    Expect(fd2,udpFrame,NULL);

    //set rule to accept frames having source ip addresses 10.1.x.x
    system("ssh -1 142.104.101.147 iptables -A FORWARD -s 10.1.1/16 -j ACCEPT");

    //send udp frame with source ip address 10.1.0.2 -- should get thru this time
    sendFrame(fd1,udpFrame,FRAMELENGTH);
    Expect(fd2,udpFrame,&UDPCompare);
}

//tests the -d match
destinationAddressTest()
{
unsigned char udpFrame[1514];
struct iphdr* ip = (struct iphdr*)(udpFrame + sizeof(struct ethhdr));

initUDPEth1(udpFrame);
initIPTables();

//set rule to accept frames having destination ip address 10.2.0.2
system("ssh -l 142.104.101.147 iptables -A FORWARD -d 10.2.0.2 -j ACCEPT");

//send udp frame with destination ip address 10.2.0.2 -- should get thru
sendFrame(fd1,udpFrame,FRAMELENGTH);
Expect(fd2,udpFrame,&UDPCompare);

//change destination ip address of udpFrame to 10.2.0.3 -- should not get thru
memcpy(&(ip->dad),IP3,4);
setIPChecksum(ip);
sendFrame(fd1,udpFrame,FRAMELENGTH);
Expect(fd2,udpFrame,NULL);

//set rule to accept frames having destination ip addresses 10.2.0.x
system("ssh -l 142.104.101.147 iptables -A FORWARD -d 10.2.0.2/24 -j ACCEPT");

//send udp frame with destination ip address 10.2.0.3 -- should get thru this time
sendFrame(fd1,udpFrame,FRAMELENGTH);
Expect(fd2,udpFrame,&UDPCompare);
}

//test the -f match
fragmentTest()
{
unsigned char udpFrame[1514];
struct iphdr* ip = (struct iphdr*)(udpFrame + sizeof(struct ethhdr));
struct udphdr* udp = (struct udphdr*)(udpFrame + sizeof(struct ethhdr) +
    sizeof(struct iphdr));
initIPTables();

// set up rule to allow all fragmented frames
system("ssh -1 142.104.101.147 iptables -A FORWARD -f -j ACCEPT");

initUDPEth1(udpFrame);

// send udp frame which is at frag_off 0 and has MF set -- not get thru
// set MF = 1 and frag_off = 0
ip->frag_off = htons(0x2000);
setIPcksum(ip);
sendDate(fd1,udpFrame,FRAMELENGTH);
Expect(fd2,udpFrame,NULL);

// send udp frame with same id with frag_off = 4 and MF unset -- get thru
ip->id = htons(1);
ip->frag_off = htons(0x0004);
memcpy(&(ip->saddr),IP4,4);
setIPcksum(ip);
sendDate(fd1,udpFrame,FRAMELENGTH);
Expect(fd2,udpFrame,&UDPCompare);
}

// tests source and destination UDP ports
UDP()
{
  unsigned char frame[1514];
  struct udphdr* udp = (struct udphdr*)(frame + sizeof(struct ethhdr) +
      sizeof(struct iphdr));

  initUDPEth1(frame);

  initIPTables();
//set rule to allow udp frames with source port 2000
system("ssh -l 142.104.101.147 iptables -A FORWARD -p udp
--sport 2000 -j ACCEPT");

//send udp frame with source port 2000 on eth1 -- should get thru
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&UDPCompare);

//send udp frame with source port 2001 on eth1 -- should not get thru
udp->source = htons(2001);
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,NULL);

//set rule to allow udp frames with a source port between 2000 and 2005
system("ssh -l 142.104.101.147 iptables -A FORWARD -p udp
--sport 2000:2005 -j ACCEPT");

//send udp frame with source port 2001 on eth1 -- should get thru this time
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&UDPCompare);

/**************DESTINATION PORT TESTS**************/

initTCPEth1(frame);
initIPTables();

//set rule to allow udp frames with destination port 2001
system("ssh -l 142.104.101.147 iptables -A FORWARD -p udp
--dport 2001 -j ACCEPT");

//send udp frame with destination port 2001 on eth1 -- should get thru
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&UDPCompare);
//send udp frame with destination port 2002 on eth1 -- should not get thru
udp->dest = htons(2002);
sendFrame(fd1, frame, FRAMELENGTH);
Expect(fd2, frame, NULL);

//set rule to allow udp frames with a destination port between 2000 and 2005
system("ssh -1 142.104.101.147 iptables -A FORWARD -p udp
--dport 2000:2005 -j ACCEPT");

//send udp frame with destination port 2002 on eth1 -- should get thru this time
sendFrame(fd1, frame, FRAMELENGTH);
Expect(fd2, frame, &UDPCompare);
}

//test extensions for tcp frames
TCP()
{
    unsigned char frame[1514];
    struct tcphdr* tcp = (struct tcphdr*)(frame + sizeof(struct ethhdr) +
        sizeof(struct iphdr));

    initTCPeth1(frame);
    initIPTables();

    //***************SOURCE PORT TESTS**************************/

    //set rule to allow tcp frames with source port 2000
    system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp
            --sport 2000 -j ACCEPT");

    //send tcp frame with source port 2000 on eth1 -- should get thru
    sendFrame(fd1, frame, FRAMELENGTH);
    Expect(fd2, frame, &TCPCompare);
//send tcp frame with source port 2001 on eth1 -- should not get thru

tcp->source = htons(2001);
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,NULL);

//set rule to allow tcp frames with a source port between 2000 and 2005
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp
   --sport 2000:2005 -j ACCEPT");

//send tcp frame with source port 2001 on eth1 -- should get thru this time
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&TCPCompare);

/******************DESTINATION PORT TESTS*****************/

initTCPEth1(frame);

initIPTables();

//set rule to allow tcp frames with destination port 2001
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp
   --dport 2001 -j ACCEPT");

//send tcp frame with destination port 2001 on eth1 -- should get thru
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&TCPCompare);

//send tcp frame with destination port 2002 on eth1 -- should not get thru
tcp->dest = htons(2002);
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,NULL);

//set rule to allow tcp frames with a dest port between 2000 and 2005
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp
   --dport 2000:2005 -j ACCEPT");
//send tcp frame with dest port 2002 on eth1 -- should get thru this time
sendFrame(fd1, frame, FRAMELENGTH);
Expect(fd2, frame, &TCPCompare);

/******************TCP FLAGS TESTS******************/

/***SYN***/
initTCPEth1(frame);
initIPTables();

//set rule to allow tcp frames with the SYN flag set and all other flags unset
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp
    --tcp-flags ALL SYN -j ACCEPT");

//set SYN bit and send tcp frame on eth1 -- should get thru
tcp->syn = 1;
sendFrame(fd1, frame, FRAMELENGTH);
Expect(fd2, frame, &TCPCompare);

//set all other bits on and SYN off -- should not get thru
tcp->syn = 0;
tcp->ack = 1;
tcp->rst = 1;
tcp->urg = 1;
tcp->fin = 1;
tcp->psh = 1;
sendFrame(fd1, frame, FRAMELENGTH);
Expect(fd2, frame, NULL);

/***ACK***/

initTCPEth1(frame);
//set rule to allow tcp frames with the ACK flag set and all other flags unset
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp
    --tcp-flags SYN,ACK ACK -j ACCEPT");

//set ACK bit and send tcp frame on eth1 -- should get thru
tcp->ack = 1;
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&TCPCompare);

//set all other bits on and ACK off -- should not get thru
tcp->ack = 0;
tcp->syn = 1;
tcp->rst = 1;
tcp->urg = 1;
tcp->fin = 1;
tcp->psh = 1;
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,NULL);

/***RST***/
initTCPeth1(frame);
initIPTables();

//set rule to allow tcp frames with the RST flag set and all other flags unset
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp
    --tcp-flags ALL RST -j ACCEPT");

//set RST bit and send tcp frame on eth1 -- should get thru
tcp->rst = 1;
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&TCPCompare);
//set all other bits on and RST off -- should not get thru
        tcp->rst = 0;
tcp->syn = 1;
tcp->ack = 1;
tcp->urg = 1;
tcp->fin = 1;
tcp->psh = 1;
sendFrame(fd1,frame,FRAMELENGTH);
        Expect(fd2,frame,NULL);

        /****FIN****/

        initTCPEth1(frame);

        initIPTables();

//set rule to allow tcp frames with the FIN flag set and all other flags unset
        system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp
                --tcp-flags ALL FIN -j ACCEPT");

//set RST bit and send tcp frame on eth1 -- should get thru
        tcp->fin = 1;
sendFrame(fd1,frame,FRAMELENGTH);
        Expect(fd2,frame,&TCPCompare);

//set all other bits on and FIN off -- should not get thru
        tcp->fin = 0;
tcp->syn = 1;
tcp->ack = 1;
tcp->urg = 1;
tcp->rst = 1;
tcp->psh = 1;
sendFrame(fd1,frame,FRAMELENGTH);
        Expect(fd2,frame,NULL);
//set rule to allow tcp frames with the URG flag set and all other flags unset
system("ssh -i 142.104.101.147 iptables -A FORWARD -p tcp
--tcp-flags ALL URG -j ACCEPT");

//set URG bit and send tcp frame on eth1 -- should get thru
tcp->urg = 1;
sendFrame(fd1, frame, FRAMELENGTH);
Expect(fd2, frame, &TCPCompare);

//set all other bits on and URG off -- should not get thru
tcp->urg = 0;
tcp->syn = 1;
tcp->ack = 1;
tcp->fin = 1;
tcp->rst = 1;
tcp->psh = 1;
sendFrame(fd1, frame, FRAMELENGTH);
Expect(fd2, frame, NULL);

//set rule to allow tcp frames with the PSH flag set and all other flags unset
system("ssh -i 142.104.101.147 iptables -A FORWARD -p tcp
--tcp-flags ALL PSH -j ACCEPT");
//set PSH bit and send tcp frame on eth1 -- should get thru
tcp->psh = 1;
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&TCPCompare);

//set all other bits on and PSH off -- should not get thru
tcp->psh = 0;
tcp->syn = 1;
tcp->ack = 1;
tcp->fin = 1;
tcp->rst = 1;
tcp->urg = 1;
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,NULL);

/***********************TCP SYN TESTS**********************/
initTCPEth1(frame);
initIPTables();

//set rule to allow syn frames
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp --syn -j ACCEPT");

//set SYN bit and send tcp frame on eth1 -- should get thru
tcp->syn = 1;
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&TCPCompare);

//set ack bit as well and send tcp frame on eth1 -- should not get thru
tcp->ack = 1;
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,NULL);
//set fin bit as well as the ack bit and send tcp frame on eth1
//-- should not get thru
tcp->fin = 1;
tcp->ack = 1;
tcp->syn = 0;
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,NULL);
}

//tests the source mac address
MAC()
{
    unsigned char udpFrame[1514];

    initUDPEth1(udpFrame);
    initIPTables();

    //set rule to accept udp frames with source MAC address 00:00:00:00:00:01
    system("ssh -1 142.104.101.147 iptables -A FORWARD -m mac
             --mac-source 00:00:00:00:00:01 -j ACCEPT");

    //send frame with source MAC 00:00:00:00:00:01 on eth1 -- should get thru
    sendFrame(fd1,udpFrame,FRAMELENGTH);
    Expect(fd2,udpFrame,&UDPCompare);

    initUDPEth2(udpFrame);

    //send frame with source MAC 00:00:00:00:00:02 on eth2 -- should not get thru
    sendFrame(fd1,udpFrame,FRAMELENGTH);
    Expect(fd2,udpFrame,NULL);
}

//test multiport extension
Multiport();
{
    unsigned char frame[1514];
    struct udphdr* udp = (struct udphdr*)(frame + sizeof(struct ethhdr) + sizeof(struct iphdr));

    initUDPEth1(frame);

    initIPTables();

    //set rule to allow udp frames with source port 2000 or 2002
    system("ssh -1 142.104.101.147 iptables -A FORWARD -m multiport --source-port 2000,2002 -j ACCEPT");

    //send udp frame with source port 2000 on eth1 -- should get thru
    sendFrame(fd1,frame,FRAMELENGTH);
    Expect(fd2,frame,&UDPCompare);

    //send udp frame with source port 2002 on eth1 -- should get thru
    udp->source = htons(2002);
    sendFrame(fd1,frame,FRAMELENGTH);
    Expect(fd2,frame,&UDPCompare);

    //send udp frame with source port 2001 on eth1 -- should not get thru
    udp->source = htons(2001);
    sendFrame(fd1,frame,FRAMELENGTH);
    Expect(fd2,frame,NULL);

    /************DESTINATION PORT TESTS************/

    initTCPEth1(frame);

    initIPTables();

    //set rule to allow udp frames with destination port 2001 or 2002
    system("ssh -1 142.104.101.147 iptables -A FORWARD -m multiport --destination-port 2001,2002 -j ACCEPT");
--destination-port 2001,2002 -j ACCEPT

//send udp frame with destination port 2001 on eth1 -- should get thru
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&UDPCompare);

//send udp frame with destination port 2002 on eth1 -- should get thru
udp->dest = htons(2002);
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,&UDPCompare);

//send udp frame with destination port 2000 on eth1 -- should not get thru
udp->dest = htons(2000);
sendFrame(fd1,frame,FRAMELENGTH);
Expect(fd2,frame,NULL);
}

//UDP connection tracking
udpStateTests()
{
    unsigned char frame1[1514];
    unsigned char tcpFrame[1514];
    struct udphdr* udp1 = (struct udphdr*)(frame1 + sizeof(struct ethhdr) +
        sizeof(struct iphdr));
    struct iphdr* ip1 = (struct iphdr*)(frame1 + sizeof(struct ethhdr));

    unsigned char frame2[1514];
    struct udphdr* udp2 = (struct udphdr*)(frame2 + sizeof(struct ethhdr) +
        sizeof(struct iphdr));

    initIPTables();

    initUDPEth1(frame1);
    initUDPEth2(frame2);
    initTCPEth2(tcpFrame);
//set rules to allow new frames that leave on eth2 and
//established frames in both directions
system("ssh -1 142.104.101.147 iptables -A FORWARD -p udp -o eth2
    -m state --state NEW -j ACCEPT");
system("ssh -1 142.104.101.147 iptables -A FORWARD -p udp -m state
    --state ESTABLISHED -j ACCEPT");

**********Session ID**********

//send out a UDP frame on eth1 -- should get thru
sendFrame(fd1,frame1,FRAMELENGTH);
Expect(fd2,frame1,&UDPCompare);
sleep(25);

//send out swapped UDP frame on eth2 -- should get thru
sendFrame(fd2,frame2,FRAMELENGTH);
Expect(fd1,frame2,&UDPCompare);

//send out a UDP frame that is not a swap on eth2 -- should not get thru
udp2->dest = htons(2002);
sendFrame(fd2,frame2,FRAMELENGTH);
Expect(fd1,frame2,NULL);

//send out swapped TCP frame on eth2 -- should get thru
sendFrame(fd2,tcpFrame,FRAMELENGTH);
Expect(fd1,tcpFrame,NULL);
sleep(35);

//send out swapped UDP frame on eth2 -- should not get thru
udp2->dest = htons(2000);
sendFrame(fd2,frame2,FRAMELENGTH);
Expect(fd1,frame2,NULL);
Assured status timeout***********/

//check for ASSURED status on connection -- timeout changes to 180 seconds
//send out a UDP frame on eth1 -- should get thru
sendFrame(fd1,frame1,FRAMELENGTH);
Expect(fd2,frame1,&UDPCompare);

//send out a swapped UDP frame on eth2 -- should get thru
sendFrame(fd2,frame2,FRAMELENGTH);
Expect(fd1,frame2,&UDPCompare);

//send out UDP frame on eth1 - this makes the connection ASSURED
sendFrame(fd1,frame1,FRAMELENGTH);
Expect(fd2,frame1,&UDPCompare);

sleep(175);

//send swapped frame on eth2 -- should get thru
sendFrame(fd2,frame2,FRAMELENGTH);
Expect(fd1,frame2,&UDPCompare);

sleep(185);

//send swapped frame on eth2 -- should not get thru
sendFrame(fd2,frame2,FRAMELENGTH);
Expect(fd1,frame2,NULL);
}

makeTCPConnection()
{
unsigned char frame1[1514];
struct tcphdr* tcp = (struct tcphdr*)(frame1 + sizeof(struct ethhdr) +
    sizeof(struct iphdr));
unsigned char frame2[1514];
struct tcphdr* tcp2 = (struct tcphdr*)(frame2 + sizeof(struct ethhdr) +
    sizeof(struct iphdr));

initTCPEth1(frame1);
initTCPEth2(frame2);

//send out TCP frame with SYN set on eth1 -- should get thru
tcp1->syn = 1;
sendFrame(fd1, frame1, FRAMELENGTH);
Expect(fd2, frame1, &TCPCompare);

//send out TCP frame with SYN and ACK sent on eth2 -- should get thru
tcp2->syn = 1;
tcp2->ack = 1;
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);

//send out TCP frame on eth1 with ACK set and SYN unset -- should get thru
tcp1->ack = 1;
tcp1->syn = 0;
sendFrame(fd1, frame1, FRAMELENGTH);
Expect(fd2, frame1, &TCPCompare);
}

tcpStateTests()
{
    unsigned char frame1[1514];
    struct tcphdr* tcp1 = (struct tcphdr*)(frame1 + sizeof(struct ethhdr) +
    sizeof(struct iphdr));

    unsigned char frame2[1514];
    struct tcphdr* tcp2 = (struct tcphdr*)(frame2 + sizeof(struct ethhdr) +
    sizeof(struct iphdr));
unsigned char udpFrame[1514];

initIPTables();

initTCPeth1(frame1);
initTCPeth2(frame2);

//set rules to allow new frames that leave on eth1 and allow established frames
//in both directions.
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp -o eth2 -m state
--state NEW --j ACCEPT");
system("ssh -1 142.104.101.147 iptables -A FORWARD -p tcp -m state --state
ESTABLISHED --j ACCEPT");

/**********Session tests with eth1 sending RST***********/

makeTCPConnection();

//send swapped frame on eth2 -- should get thru
tcp2->ack = 1;
sendFrame(fd2,frame2,FRAMELENGTH);
Expect(fd1,frame2,&TCPCompare);

//send not swapped frame on eth2 -- should not get thru
tcp2->source = htons(2002);
sendFrame(fd2,frame2,FRAMELENGTH);
Expect(fd1,frame2,NULL);

//send out UDP frame with swapped id on eth2 -- should not get thru
initUDPeth2(udpFrame);
sendFrame(fd2,udpFrame,FRAMELENGTH);
Expect(fd1,udpFrame,NULL);

//send out RST on eth1 -- should get thru -- status changes to CLOSE
tcp1->rst = 1;
sendFrame(fd1, frame1, FRAMELENGTH);
Expect(fd2, frame1, &TCPCompare);

sleep(5);

initTCPEth2(frame2);
tcp2->ack = 1;
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);

sleep(15);

sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, NULL);

/***Session tests with eth2 sending RST*************/

makeTCPConnection();

//send out RST on eth2 -- should get thru -- changes to CLOSE
tcp2->rst = 1;
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);

sleep(5);

initTCPEth2(frame2);
tcp2->ack = 1;
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);

sleep(15);

sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, NULL);
/*********** Normal Termination started by eth2 ***********/

makeTCPConnection();

// send out FIN + ACK on eth2 -- should get thru -- changes to CLOSE WAIT
initTCPEth2(frame2);
tcp2->fin = 1;
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);

// send out ACK on eth1 -- should get thru
initTCPEth1(frame1);
tcp1->ack = 1;
sendFrame(fd1, frame1, FRAMELENGTH);
Expect(fd2, frame1, &TCPCompare);

// send out FIN + ACK on eth1 -- should get thru -- changes to TIME WAIT
tcp1->fin = 1;
sendFrame(fd1, frame1, FRAMELENGTH);
Expect(fd2, frame1, &TCPCompare);

// send out ACK on eth2 -- should get thru
tcp2->ack = 1;
tcp2->fin = 0;
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);
sleep(115);

// now send frame on eth2 -- should get thru
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);
sleep(125);
//now send frame on eth2 -- should not get thru
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, NULL);

/************************Normal Termination started by eth1**************/
makeTCPConnection();

//send out FIN + ACK on eth1 -- should get thru -- changes to FIN_WAIT
initTCPEth1(frame1);
tcp1->fin = 1;
tcp1->ack = 1;
sendFrame(fd1, frame1, FRAMELENGTH);
Expect(fd2, frame1, &TCPCompare);

//send out ACK on eth2 -- should get thru
initTCPEth2(frame2);
tcp2->ack = 1;
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);

//send out FIN + ACK on eth2 -- should get thru -- changes to TIME WAIT
tcp2->fin = 1;
tcp2->ack = 1;
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);

//send out ACK on eth1 -- should get thru
tcp1->ack = 1;
tcp1->fin = 0;
sendFrame(fd1, frame1, FRAMELENGTH);
Expect(fd2, frame1, &TCPCompare);
sleep(115);
//now send frame on eth2 -- should get thru
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, &TCPCompare);

sleep(125);

//now send frame on eth2 -- should not get thru
sendFrame(fd2, frame2, FRAMELENGTH);
Expect(fd1, frame2, NULL);
}

stateTests()
{
    udpStateTests();

    tcpStateTests();
}

//test SNAT targets
SNAT()
{
    unsigned char frame1[1514];
    unsigned char frame2[1514];
    int received = -1;

    struct iphdr* ip1 = (struct iphdr*)(frame1 + sizeof(struct ethhdr));
    struct udphdr* udp1 = (struct udphdr*)(frame1 + sizeof(struct ethhdr) +
                                sizeof(struct iphdr));

    struct iphdr* ip2 = (struct iphdr*)(frame2 + sizeof(struct ethhdr));
    struct udphdr* udp2 = (struct udphdr*)(frame2 + sizeof(struct ethhdr) +
                                sizeof(struct iphdr));

    initIPTables();
initNATTable();

initUDPEth1(frame1);
initUDPEth2(frame2);

//Translate the source IP address to 10.2.0.3
system("ssh -1 142.104.101.147 iptables -t nat -A POSTROUTING -o eth2 -j SNAT --to-source 10.2.0.3");

//Send frame on eth1 with source IP 10.1.0.1
sendFrame(fd1,frame1,FRAMELENGTH);

//Send out frame on eth2 with destination IP address 10.2.0.3, Expect frame
//with dest IP 10.1.0.1
memcpy(&(ip2->dad&),IP5,4);
setIPCksum(ip2);
sendFrame(fd2,frame2,FRAMELENGTH);

initUDPEth2(frame2);
Expect(fd1,frame2,&UDPCompare);

initIPTables();
initNATTable();

initUDPEth1(frame1);
initUDPEth2(frame2);

//Translate source IP to 10.2.0.4 or 10.2.0.5 and source port to 15000
system("ssh -1 142.104.101.147 iptables -t nat -A POSTROUTING -o eth2 -j SNAT --to-source 10.2.0.4-10.2.0.5:15000");

//Send frames with source IP 10.1.0.1 and 10.1.0.2
sendFrame(fd1,frame1,FRAMELENGTH);
memcpy(&(ip1->saddr),IP4,4);
setIPCksum(ip1);
sendFrame(fd1,frame1,FRAMELENGTH);

//Send frames with dest IP 10.2.0.4 and 10.2.0.5 and destination port 15000,
//expect frames with dest IP 10.1.0.1 and 10.1.0.2 and dest port 2000
//10.2.0.4
memcpy(&(ip2->daddr),IP6,4);
udp2->dest = htons(15000);
setIPcksum(ip2);
sendFrame(fd2,frame2,FRAMELENGTH);

initUDPEth2(frame2);
if(!Expect(fd1,frame2,&UDPCompare)) {
    memcpy(&(ip2->daddr),IP4,4);
    Expect(fd1,frame2,&UDPCompare)
    received = 2;
}
else
    received = 1;

//10.2.0.5
memcpy(&(ip2->daddr),IP7,4);
udp2->dest = htons(15000);
setIPcksum(ip2);
sendFrame(fd2,frame2,FRAMELENGTH);

initUDPEth2(frame2);
if(received == 2)
    Expect(fd1,frame2,&UDPCompare);
else {
    memcpy(&(ip2->daddr),IP4,4);
    Expect(fd1,frame2,&UDPCompare)
}

//Test MASQUERADE target
MASQ()
{
    unsigned char frame1[1514];
    unsigned char frame2[1514];

    struct iphdr* ip1 = (struct iphdr*)(frame1 + sizeof(struct ethhdr));
    struct udphdr* udp1 = (struct udphdr*)(frame1 + sizeof(struct ethhdr) +
            sizeof(struct iphdr));

    struct iphdr* ip2 = (struct iphdr*)(frame2 + sizeof(struct ethhdr));
    struct udphdr* udp2 = (struct udphdr*)(frame2 + sizeof(struct ethhdr) +
            sizeof(struct iphdr));

    int received = -1;

    initIPTables();
    initNATTTable();

    initUDPEth1(frame1);
    initUDPEth2(frame2);

    // Set rule to masquerade frames that leave on eth2
    system("ssh -1 142.104.101.147 iptables -t nat -A POSTROUTING -o eth2
            -j MASQUERADE");

    // Frame's source IP is translated to 10.2.2.2
    sendFrame(fd1, frame1, FRAMELENGTH);

    // Send frame with destination IP 10.2.2.2,
    // Expect frame with destination IP 10.1.0.1
    memcpy(&(ip2->daddr), IP11, 4);
    setIPChecksum(ip2);
    sendFrame(fd2, frame2, FRAMELENGTH);
initUDPEth2(frame2);
Expect(fd1, frame2, &UDPCompare);

initIPTables();
initNATTable();

initUDPEth1(frame1);
initUDPEth2(frame2);

// Masquerade frames leaving on eth2 and translate their source port
to 17000 or 17001
system("ssh -1 142.104.101.147 iptables -t nat -A POSTROUTING -p udp
-o eth2 -j MASQUERADE --to-ports 17000-17001");

// Send frames with source IP 10.1.0.1 and 10.1.0.2,
// they are masqueraded and their source port translated to 17000 or 17001
sendFrame(fd1, frame1, FRAMELENGTH);
memcpy(&(ipl->sadr), IP4, 4);
setIPChecksum(ip1);
sendFrame(fd1, frame1, FRAMELENGTH);

// Send frames with dest IP 10.2.2.2 and destination ports 17000 and 17001,
// expect frames with dest IP 10.1.0.1 and 10.1.0.2 and dest port 2000
// 17000
memcpy(&(ip2->daddr), IP4, 4);
udp2->dest = htons(17000);
setIPChecksum(ip2);
sendFrame(fd2, frame2, FRAMELENGTH);

initUDPEth2(frame2);
if (!Expect(fd1, frame2, &UDPCompare)) {
    memcpy(&(ip2->daddr), IP4, 4);
    Expect(fd1, frame2, &UDPCompare)
    received = 2;
}
else
    received = 1;

//17001
memcpy(&(ip2->daddr),IP11,4);
udp2->dest = htons(17001);
setIPCksum(ip2);
sendFrame(fd2,frame2,FRAMELENGTH);

initUDPEth2(frame2);
if (received == 2)
    Expect(fd1,frame2,&UDPCompare);
else {
    memcpy(&(ip2->daddr),IP4,4);
    Expect(fd1,frame2,&UDPCompare)
}

//test DNAT targets
DNAT()
{
    unsigned char frame1[1514];
    unsigned char frame2[1514];
    int received = -1;

    struct iphdr* ip1 = (struct iphdr*)(frame1 + sizeof(struct ethhdr));
    struct udphdr* udp1 = (struct udphdr*)(frame1 + sizeof(struct ethhdr) +
        sizeof(struct iphdr));

    struct iphdr* ip2 = (struct iphdr*)(frame2 + sizeof(struct ethhdr));
    struct udphdr* udp2 = (struct udphdr*)(frame2 + sizeof(struct ethhdr) +
        sizeof(struct iphdr));

    initIPTables();
    initNATTable();
initUDPEth1(frame1);
initUDPEth2(frame2);

//Translate the destination IP address to 10.1.0.3
system("ssh -1 142.104.101.147 iptables -t nat -A PREROUTING -i eth2 -p udp --dport 2000 -j DNAT --to-destination 10.1.0.3");

//Send frame on eth2 with destination IP 10.1.0.1, frame's destination IP
//is translated to 10.1.0.3
sendFrame(fd2,frame2,FRAMELENGTH);

//Send out frame on eth1 with source IP address 10.1.0.3, Expect frame with
//source IP 10.1.0.1
memcpy(&(ip1->saddr),IP8,4);
setIPCKsum(ip1);
sendFrame(fd1,frame1,FRAMELENGTH);

initUDPEth1(frame1);
Expect(fd2,frame1,UDPCompare);

initIPTables();
initNATTable();

initUDPEth1(frame1);
initUDPEth2(frame2);

//Translate destination IP to 10.1.0.4 or 10.1.0.5 and destination port to 15000
system("ssh -1 142.104.101.147 iptables -t nat -A PREROUTING -i eth2 -p udp --dport 2000 -j DNAT --to-destination 10.1.0.4:10.1.0.5:15000");

//Send frames with destination IP 10.1.0.1 and 10.1.0.2
sendFrame(fd2,frame2,FRAMELENGTH);
memcpy(&(ip2->daddr),IP4,4);
setIPCKsum(ip2);
sendFrame(fd2, frame2, FRAMELENGTH);

// Send frames with source IP 10.1.0.4 and 10.1.0.5 and source port 15000,
// expect frames with source IP 10.1.0.1 and 10.1.0.2 and source port 2000
// 10.1.0.4
memcpy(&(ip1->saddr), IP9, 4);
udp1->source = htons(15000);
setIPChecksum(ip1);
sendFrame(fd1, frame1, FRAMELENGTH);

initUDPEth1(frame1);
if(!Expect(fd2, frame1, &UDPCompare)) {
    memcpy(&(ip1->saddr), IP4, 4);
    Expect(fd2, frame1, &UDPCompare)
    received = 2;
}
else
    received = 1;

// 10.1.0.5
memcpy(&(ip1->saddr), IP10, 4);
udp1->source = htons(15000);
setIPChecksum(ip1);
sendFrame(fd1, frame1, FRAMELENGTH);

initUDPEth1(frame1);
if(received = 2)
    Expect(fd2, frame1, &UDPCompare);
else {
    memcpy(&(ip1->saddr), IP4, 4);
    Expect(fd2, frame1, &UDPCompare)
}

main(int argc, char* argv[])


```c
{
    //create and bind inbound and outbound sockets
    fd1 = ss_socket();
    ss_bind(fd1,"eth1");

    fd2 = ss_socket();
    ss_bind(fd2,"eth2");

    inInterfaceTest();
    outInterfaceTest();
    protocolTest();
    sourceAddressTest();
    destinationAddressTest();
    fragmentTest();
    UDP();
    TCP();
    MAC();
    Multiport();
    stateTests();
    SNAT();
    MASQ();
    DNAT();
```
//close fd1 and fd2
ss_close(fd1);
ss_close(fd2);
}

D.0.8  limitTest.c
#include<stdio.h>
#include "socketLibrary.h"
#include "headerUtility.h"

#define FRAMELENGTH 64

const int IPHDRLEN = 20;

//constants for addresses and ports
const unsigned char MAC1[6] = {0x00,0x00,0x00,0x00,0x00,0x01};
const unsigned char MAC2[6] = {0x00,0x50,0xBF,0x91,0xC0,0xA5};
const unsigned char IP1[4] = {10,1,0,1};
const unsigned char IP2[4] = {10,2,0,2};
const unsigned short PORT1 = 2000;
const unsigned short PORT2 = 2001;

int fd1,fd2,i;

//compare 2 udp frames and return 1 if they are identical and 0 if different
int UDPCompare(void* expectedFrame,void* actualFrame)
{
    struct iphdr actualIPHdr;
    struct iphdr expectedIPHdr;
    struct udphdr actualUDPHdr;
    struct udphdr expectedUDPHdr;
getUDPHeader(&expectedUDPHdr, getIPHeader(&expectedIPHdr, expectedFrame + 14));
getUDPHeader(&actualUDPHdr, getIPHeader(&actualIPHdr, actualFrame));

if ((actualIPHdr.saddr == expectedIPHdr.saddr) &&
    (actualIPHdr.daddr == expectedIPHdr.daddr) &&
    (actualIPHdr.protocol == expectedIPHdr.protocol) &&
    (expectedUDPHdr.source == actualUDPHdr.source) &&
    (expectedUDPHdr.dest == actualUDPHdr.dest))
    return(1);
else
    return(0);
}

// expect a frame on fd and return 1 if expected frame arrives or 0 if
// expected frame does not arrive. No frame is expected when compare = NULL
int Expect(int fd, void* expectedFrame, int (*compare)(void*, void*))
{
    struct ethhdr ethhdr;
    unsigned char buffer[1514];

    if (compare == NULL) {
        if (ss_receive(fd, &ethhdr, buffer, 5) > 0)
            return(0);
        else
            return(1);
    } else {
        if (ss_receive(fd, &ethhdr, buffer, 5) <= 0)
            return(0);
        else {
            if ((*compare)(expectedFrame, buffer))
                return(1);
            else
                return(0);
        }
    }
}
//flush all iptables rules and set the FORWARD policy to drop
void initIPTables()
{
    //flush all rules
    system("ssh -1 142.104.101.147 iptables -F");

    //set FORWARD policy to DROP
    system("ssh -1 142.104.101.147 iptables --policy FORWARD DROP");
}

//tests --limit and --limit-burst
void limitTest()
{
    unsigned char frame[60];
    struct ethhdr* ethhdr = (struct ethhdr*)frame;
    struct iphdr* iphdr = (struct iphdr*)(frame + sizeof(struct ethhdr));
    struct udphdr* udphdr =
        (struct udphdr*)((frame + sizeof(struct ethhdr)) + sizeof(struct iphdr));

    system("ssh -1 142.104.101.147 iptables -A FORWARD -o eth2 -p udp -m limit
            --limit 1/second --limit-burst 3 -j ACCEPT");

    ss_setEthernetHeader(ethhdr,MAC2,MAC1,ETH_P_IP);
    setIPHeader(iphdr,IP2,IP1,IPPROTO_UDP,FRAMELENGTH);
    setUDPHdr(udphdr,PORT2,PORT1,FRAMELENGTH);

    // Send a burst of --limit-burst frames; all frames accepted
    for(i=0;i<3;i++)
    {
        ss_send(fd1,frame,60);
        Expect(fd2,frame,&UDPCompare);
    }

    // Send a frame; it should be dropped
    ss_send(fd1,frame,60);
Expect(fd2,frame,NULL);

// Pause long enough to fill the token bucket
usleep(3000000);
// Send a burst of --limit-burst frames; all frames accepted
for(i=0;i<3;i++) {
    ss_send(fd1,frame,60);
    Expect(fd2,frame,&UDPCompare);
}
// Pause to accumulate a token
usleep(1000000);
// Send a frame; it should be accepted
ss_send(fd1,frame,60);
Expect(fd2,frame,&UDPCompare);
}

main(int argc,char* argv[])
{
    fd1 = ss_socket();
    fd2 = ss_socket();
    ss_bind(fd1,"eth1");
    ss_bind(fd2,"eth2");

    initIPTables();

    limitTest();

    ss_close(fd1);
    ss_close(fd2);
}

D.0.9 performanceTest.c
#include<stdio.h>
#include "socketLibrary.h"
#include "headerUtility.h"

// constants for addresses and ports
const unsigned char MAC1[6] = {0x00,0x00,0x00,0x00,0x00,0x01};
const unsigned char MAC2[6] = {0x00,0x50,0xBF,0x91,0xC0,0xA5};

const unsigned char IP1[4] = {10,1,0,1};
const unsigned char IP2[4] = {10,2,0,2};

const unsigned short PORT1 = 2000;
const unsigned short PORT2 = 2001;

int totalUsec = 0;
int totalSec = 0;

int framelengths[] = {64,128,256,512,1024,1280,1518};

void tvsub(struct timeval* tv0,struct timeval* tv1)
{
    if(tv0->tv_usec < tv1->tv_usec){
        totalUsec = totalUsec + tv0->tv_usec - tv1->tv_usec + 1000000;
        totalSec = totalSec + tv0->tv_sec - tv1->tv_sec - 1;
    }
    else {
        totalUsec = totalUsec + tv0->tv_usec - tv1->tv_usec;
        totalSec = totalSec + tv0->tv_sec - tv1->tv_sec;
    }
}

main(int argc,char* argv[])
{
    int fd1,fd2,j,i;

    unsigned char* frame;
    unsigned char* frame1;
struct ethhdr* ethhdr;
struct iphdr* iphdr;
struct udphdr* udphdr;

struct timeval tv;
struct timeval tv0;
struct timeval* tvl;

fd1 = ss_socket();
fd2 = ss_socket();
ss_bind(fd1,"eth1");
ss_bind(fd2,"eth2");

frame = (unsigned char*)malloc(framelengths[i]);
frame1 = (unsigned char*)malloc(framelengths[i]);

ethhdr = (struct ethhdr*)frame;
iphdr = (struct iphdr*)(frame + sizeof(struct ethhdr));
udphdr = (struct udphdr*)(frame + sizeof(struct ethhdr) + sizeof(struct iphdr));

for(i=0;i<7;i++) {
    ss_setEthernetHeader(ethhdr,MAC2,MAC1,ETH_P_IP);
    setIPHeader(iphdr,IP2,IP1,IPPROTO_UDP,i);
    setUDPHdr(udphdr,PORT2,PORT1,i);

    for(j=0;j<10;j++) {
        usleep(25000);
        gettimeofday((struct timeval*)(frame+framelengths[i]-12),NULL);
        ss_send(fd1,frame,framelengths[i]-4);

        if(read(fd2,frame1,framelengths[i]-4) > 0) {
            tv1 = (struct timeval*)(frame1+framelengths[i]-12);
            gettimeofday(&tv,NULL);
            tvsub(tv,tv1);
printf("avg usec = %d\n", totalUsec/10);
printf("avg sec = %d\n", totalSec/10);

totalSec = 0;
totalUsec = 0;
}
ss_close(fd1);
ss_close(fd2);