Advanced Packet Obfuscation and Control program (Apoc)

Adam Lee Hahn
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Advanced Packet Obfuscation and Control program (Apoc)

by

Adam Lee Hahn

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Co-major: Computer Engineering; Information Assurance

Program of Study Committee:
Doug Jacobson, Major Professor
Cliff Bergman
Tom Daniels

Iowa State University
Ames, Iowa
2006

Copyright © Adam Lee Hahn, 2006. All rights reserved.
Graduate College
Iowa State University

This is to certify that the Master’s thesis of
Adam Lee Hahn
has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

LIST OF FIGURES ......................................................... vi

ABSTRACT ........................................................................ viii

CHAPTER 1 Introduction .................................................. 1

CHAPTER 2 Background ................................................... 4

Internet Basics ................................................................. 4

Physical Layer ................................................................. 5

Data Link Layer ............................................................... 5

Network Layer ................................................................. 7

Transport Layer .............................................................. 9

Session, Presentation, and Application Layer .................. 10

ISEAGE ................................................................. 11

How ISEAGE Works ....................................................... 12

Ettercap ................................................................. 14

How Ettercap Works ...................................................... 14

Netfilter ................................................................. 15

How Netfilter Works ...................................................... 15

CHAPTER 3 Practical Uses of Apoc ................................. 18

Randomization .............................................................. 18

MAC address modification ............................................. 20
# LIST OF FIGURES

2.1 The OSI model and packet construction ............................................. 5  
2.2 An Ethernet packet ................................................................. 6  
2.3 An IPv4 header ................................................................. 8  
2.4 ARP packet ................................................................. 9  
2.5 UDP header ............................................................. 10  
2.6 TCP header ............................................................. 11  
2.7 ISEAGE’s basic structure (4) ......................................................... 13  
2.8 Netfilter connection tracking structures(6) ........................................... 17  
3.1 Bypassing switches with broadcast MACs ........................................... 21  
4.1 Apoc’s threaded structure ......................................................... 23  
4.2 block stuct ............................................................. 25  
4.3 ifblock stuct ............................................................. 26  
4.4 Apoc connection tracking structures ............................................... 32  
4.5 Packet reiteration ............................................................. 35  
4.6 Avoiding packet reiteration ......................................................... 36  
A.1 Nested if statements ........................................................... 48  
A.2 Inaccurate conditional statements ................................................. 49  
A.3 Modifying Ethernet fields ....................................................... 50  
A.4 Modifying Arp fields ........................................................... 51
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.5</td>
<td>Modifying IP fields</td>
<td>51</td>
</tr>
<tr>
<td>A.6</td>
<td>Modifying ICMP fields</td>
<td>52</td>
</tr>
<tr>
<td>A.7</td>
<td>Modifying TCP fields</td>
<td>52</td>
</tr>
<tr>
<td>A.8</td>
<td>Modifying UDP fields</td>
<td>53</td>
</tr>
</tbody>
</table>
ABSTRACT

The Advanced Packet Obfuscation and Control program (Apoc) was developed to extend the functionality of the ISEAGE lab. The Internet Scale Event and Attack Generation Environment (ISEAGE) lab implements a virtual Internet testbed where cyber attacks can be tested. Apoc is a very dynamic tool which can be configured to implement a number of modifications to packets as they traverse the network. These modifications primarily focus on increasing the flow of traffic through ISEAGE and abstracting the source of attacks. Apoc works by interpreting a user provided script which specifies the modifications to be performed. Although Apoc was inspired primarily for use in the ISEAGE lab, it is also a useful tool for many real world problems.
CHAPTER 1 Introduction

The development of strong security testing tools is necessary to provide administrators with the ability to ensure security in their systems. Recent reports by the Government Accountability Office from fiscal year 2005 report that the “testing of security controls” is a major problem in information security(1). The Advanced Packet Obfuscation and Control program (Apoc) is a tool that will provide a number of useful features for security testing purposes. The main focus in the development of Apoc was the implementation of a framework which provides the ability to perform arbitrary modifications to network traffic.

There are a number of tools currently available which focus on the modification of network traffic. The end goal of these modifications can vary greatly from changing network topology to performing malicious acts. An example of a more benign modification is network address translation (NAT), which is used to allow any number of hosts on an internal network to share one unique external address. On the other end of this spectrum are tools that provide a means to create Man in the Middle (MitM) attack. A MitM attack performs some malicious act based on the attacker’s ability to modify the traffic between two hosts.

Apoc provides a framework to accomplish a range of the goals attained by the previously mentioned tools. Apoc is a separate host that is placed in the network between communicating hosts. Therefore, all traffic between these hosts is routed through Apoc providing it complete control over this traffic. This framework attempts to deduce all desirable functionality out of its ability to control this traffic. In this scope Apoc is
focused on providing functions ranging from helping with network configuration and communication to providing a means to launch malicious attacks.

Apoc’s ability to be a central mediator of all packets on a network creates some similarities to a NAT or firewall. The functionality of Apoc differs greatly from both of these as it is not aimed at increasing the security of a network. NATs and firewalls typically utilize their centralized location to limit the passing of any unwanted traffic. Apoc’s uses are generally coupled with testing exercises and are not intended to be used for typical network operations. This introduces some significant differences between these tools.

First Apoc is focused on performing many arbitrary modifications to packets. While a NAT may have to perform general modifications to IP addresses and ports, Apoc must be able to perform very fine grain modifications. This includes changing the value of any specific fields in a packet and introducing an element of randomness to the traffic.

Secondly, as Apoc is not a typical network gateway it has significantly different requirements from a NAT or firewall. Typically a NAT or firewall will focus on filtering out any data that may be malicious. Since most of the traffic passed through Apoc is done so for security testing purposes, it must be able to accurately forward all malicious traffic. This requires that Apoc handle traffic slightly differently than a NAT or firewall.

Thirdly, Apoc is not an addressable machine. A packet will never be addressed to Apoc, as it only modifies packets destined to other hosts. Apoc’s presence on the network will be completely unnoticeable by any host on that network. Apoc also does not decrement the IP Time to Live field as a NAT or firewall would.

Apoc’s design was inspired to meet the specific needs of the Internet Scale Event and Attack Generation Environment (ISEAGE) lab. ISEAGE is an extremely scalable network testbed developed here at Iowa State University. The primary goal of ISEAGE is to recreate a realistic model of the Internet in a confined lab environment. Apoc provides a method to perform a number of modifications to ISEAGE’s network traffic
with an end goal of creating a more realistic Internet environment. Apoc assists ISEAGE in number of ways, mostly focusing on creating traffic that appears more realistic and providing a more abstract means to launch attacks. The use of Apoc extends beyond the scope of ISEAGE as many of its functions have practical uses in other situations.

Chapter 2, titled Background, will provide an overview of many technologies relevant to Apoc. Chapter 3, titled Practical Uses of Apoc, will explain how Apoc can be used. Chapter 4, titled Design and Implementation, will describe the inner workings of Apoc. Chapter 5, titled Results, will explain how Apoc has meet it’s initial goals. Chapter 6, titled Summary, will conclude the work performed on this project. Chapter 7, titled Future Work, will explain what should be accomplished if work on Apoc continues.
CHAPTER 2 Background

The development of Apoc is based upon a number of current technologies. This section focuses on introducing these technologies and describing their relevance to Apoc. It starts by explaining basic network concepts. Next it explains the ISEAGE project which inspired its creation. Finally it introduces two tools, Ettercap and Netfilter, which have had a strong influence on the development of Apoc.

Internet Basics

A thorough understanding of Apoc will require a solid background in computer networking. This section will attempt to provide a brief review of the current networking protocols and ideologies that Apoc’s functionality is based upon.

The most popular model used in describing computer networks is the Open Systems Interconnections (OSI) model. The OSI model consists of seven different layers. Each layer describes a set of requirement necessary to implement a network communication between two hosts. Although the OSI model is the most popular network model, it was developed after initial Internet research was performed. This tends to cause discrepancies between the OSI model and Internet, however, the OSI model will still be used in this review. In this section each layer of the OSI model will be explained along with any relevant protocols.

An example of the OSI model and how it is used to create and transmit packets is shown in Figure 2.1. This figure shows how a packet is created as each layer adds
its protocol headers to the front of the data received from the upper layer. First the application level data is added, then the transport layer protocol header is added to the front. The network and data link layer protocol headers are then added accordingly. Finally, all of the data is transmitted at the network interface by the physical layer.

![Figure 2.1 The OSI model and packet construction](image)

**Physical Layer**

This layer deals with the physical transmission of bits over a network. It deals mostly with the transmission of electronic signals over some physical medium whether it be a wire or radio waves. This layer is particularly concerned with things like transmission rates, voltage levels, and bit encoding. The workings of the physical layer are not particularly important in the development of Apoc.

**Data Link Layer**

The data link layer provides a means for implementing reliable data transmission across the physical network. This entails the creation of streams of bits which can be grouped together and sent to another host. These streams are typically referred to as frames. Data link protocols must provide some addressing mechanism to ensure that
each frame arrives at the desired host. These frames should also support some sort of error checking in case a frame becomes damaged during transmission. The Data Link layer is typically split into two separate layers, the Logical Link Control (LLC) layer and the Media Access Control (MAC) layer. The MAC layer sits directly on top of the physical layer and is responsible for developing the frames and working with other hosts on the network to contend for the shared physical media. The LLC layer works directly with the flow control and error control aspects of the data link layer.

Ethernet

The only data link protocol supported in Apoc is Ethernet. Ethernet is a very popular wire based protocol for local area networks. Each host on an Ethernet network has a unique MAC address. Since every MAC address is unique, each host on the local network can be accessed through its address. A MAC address is 48 bits long and is commonly expressed the following notation 01:23:45:67:89:0a. Each frame on an Ethernet network contains a destination MAC address, a source MAC address, a 16 bit type field, and some data. Typically an Ethernet interface only reads a frame if the destination MAC address of the frame matches its own MAC address. However, Ethernet interfaces can be placed into what is called promiscuous mode where it reads every frame. A MAC address of ff:ff:ff:ff:ff:ff is considered a broadcast address which means every host that receives this frame reads it. Figure 2.2 shows the structure of an Ethernet frame.

![Figure 2.2 An Ethernet packet](image)
Network Layer

The data link layer is limited to the transmission of frames on a single network. The network layer provides a means to deliver packets across multiple networks. This introduces two features, logical addresses and routing. Network layer addresses must be logical, that is, they should be allocated in a structure where the address provides information on the location of this host. Routing introduces the theory in which a packet travels throughout the Internet based solely on its logical address. The most popular protocols on the network layer are the Internet Protocol (IP) versions 4 and 6, the Address Resolution Protocol (ARP), and the Internet Control Message Protocol (ICMP). The following sections elaborate on these common network layer protocols.

IP

The IP protocol provides the basic foundation for the Internet. Currently there are two versions of the IP protocol. Version 4 (IPv4) is the version that the current Internet is based upon. IPv4 was developed years ago and is starting to show its age. Version 6 (IPv6) was created as the successor to IPv4. While there is a significant push to transition to IPv6, it is a slow and difficult task.

In IPv4 each host has a unique 32 bit address expressed in the following format 192.168.0.10. Figure 2.3 shows an IPv4 header. Each IP packet has a field for both the source and destination address. Another field of interest is the Time to Live (TTL). This specifies the number of times this packet can be routed. Each time this packet is forwarded this field is decremented. Once the field is zero the packet will not be further routed.
ICMP

The ICMP protocol was created to be used along with IP to perform many error reporting tasks. ICMP messages can be used to carry either error reporting or query messages. An IP header is prepended to an ICMP packet in order to provide the necessary addressing capabilities. Although the ICMP data sits inside an IP header, ICMP is still considered a network layer protocol as it is an extension to IP and does not perform any upper layer tasks. The most common ICMP packets are the Echo Request/Reply messages. These are typically used to test if a host is running. An Echo Request query is sent to a host, if that host is functioning properly it responds with an Echo Reply message.

ARP

Although ARP sits on the network layer it is not an inter-network protocol like IP. ARP is used to obtain an unknown MAC address with a known IP address. Whenever a host sends a frame it must know the MAC address of the host intended to receive that frame, however, often times a host will only have the IP address for the destination host. ARP works by sending a request packet to the broadcast MAC address, this ARP Request packet contains the known IP address. Each host that receives this will check
to see if its IP matches the one found in the ARP Request packet. If a match is found, that host will send an ARP Reply packet to the host that sent the Request. The ARP Reply packet contains the MAC address that matches the questioned IP address. Most operating systems keep a small dynamic cache table which maps IP address to their known MAC addresses. This prevents a host from having to perform an ARP Request every time a frame is sent. Figure 2.4 shows the format of an ARP packet.

![Figure 2.4 ARP packet](image)

**Transport Layer**

Due to the functionality of the network layer all hosts on the Internet are able to communicate with each other. However, once a packet is received there needs to be a method to map that packet to the application which will process it. The transport layer addresses this problem by implementing port numbers. A port number is a 16 bit number which is used to map an application running on a host to the packets it sends and receives. Each application has its own unique port so there are no discrepancies between packets and their intended applications. For example, port 80 is reserved for web servers so all packets sent to port 80 will be processed by the web server on that host. The two transport layer protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), these are explained below.
UDP

UDP is a very simple protocol consisting only of the source and destination port numbers, the packet length, and a checksum. It provides no means to ensure accurate delivery of the packet. Typically UDP is only used when occasional packet loss is acceptable. Figure 2.5 shows an UDP header.

<table>
<thead>
<tr>
<th>0</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>Destination Port</td>
<td></td>
</tr>
<tr>
<td>Total Length</td>
<td>Checksum</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.5 UDP header

TCP

TCP is a reliable transport layer protocol, it provides mechanisms to assure that all data is properly received. TCP is an incredibly detailed protocol and will only be briefly covered in this document. Reliability is provided by establishing a connection between communicating hosts. After this connection is established, packet delivery is ensured through Acknowledgment packets. When a host receives a packet it sends an Acknowledgment packet, or Ack, back to the sender notifying that the packet was received. Through this method a host will know if its packet was successfully received. A host can continually resend a packet until an Ack is received. Figure 2.6 shows the TCP header format.

Session, Presentation, and Application Layer

The highest three layers in the OSI model are the Session, Presentation, and Application layers. These layers represent an area where the OSI model does accurately model common Internet traffic. Most current Internet communications only use the application layer, ignoring both the Session and Presentation layer.
The session layer was created to perform any necessary management functions between communicating hosts. The Presentation layer performs various data coding strategies such as compression or encryption. It is used to make sure that each host can understand the data formats used by other hosts. The application layer is the layer that is directly accessed by the end user applications. All common Internet based applications such as email, web, or FTP have their own application layer protocol. This is typically encapsulated in either a TCP or UDP packet depending on the application layer protocol.

**ISEAGE**

The Internet Scale Event and Attack Generation Environment (ISEAGE) was developed at Iowa State University by Dr. Doug Jacobson. ISEAGE creates a virtual Internet environment designed for “researching, designing, and testing cyber defense mechanisms”(3). ISEAGE strives to create an environment that accurately portrays all characteristics of the authentic Internet. This allows researchers to test security solutions in a controlled environment while still maintaining the key attributes of the Internet. The benefits of ISEAGE extend well beyond academic research. ISEAGE has already shown that it is a valuable asset in the education of both students and professionals as
it has already been the host location for computer security camps, training sessions, and three Cyber Defense Competitions.

ISEAGE has established a set of domains where it will aid in solving many current information security problems (3). These domains are described below.

- Critical Infrastructure - ISEAGE can be used to recreate critical sections of the infrastructure which can be tested for security and survivability without the threat of damaging the actual infrastructure.

- End User Security - ISEAGE provides an ideal environment to train users on security related issues and new technologies.

- Academic and Industrial Research - Researchers are provided with an environment to test new ideas and devices.

- Forensics and Law Enforcement Agencies - ISEAGE can be used to experiment with tools attackers use and find novel ways to trace attacks.

**How ISEAGE Works**

ISEAGE’s core consists of a group of gigabit switches. Connected to these switches is a 64 node rack. Each node on this rack runs an instance of a virtual subnet. A virtual subnet is a piece of software that emulates a router on the node. Each virtual subnet is able to emulate a number of routers. The combination of all these routers connected to the core switches allows ISEAGE to mimic literally hundreds of subnets providing the ability to reproduce large sections of the Internet (4). Figure 2.7 shows the basic structure of the ISEAGE lab.

ISEAGE’s core provides the structure of the virtual Internet, but it alone will not provide an authentic environment. Internet traffic has a great deal of both malicious and non-malicious traffic traversing it at all times between millions of different hosts. To
emulate these characteristics in ISEAGE a number of tools are currently being developed. Apoc was developed as one of these tools. Other tools include traffic generators to provide ISEAGE with a great deal of realistic background traffic. There is a set of attack launching tools which allows complex attacks to be easily launched from ISEAGE. Also there are virtual hosts and networks which will provide ISEAGE with a large number of end nodes to mimic the millions of end hosts on the actual Internet.
Ettercap

Ettercap is touted as a “multipurpose sniffer/interceptor/logger for switched networks” (5). Ettercap is a very powerful tool with a great deal of functionality. It can be run as a packet sniffer, and further used to steal any passwords on the network. It also provides the user with the ability to perform a number of MitM attacks on a local network by manipulating different protocols. Once Ettercap has established a MitM attack it can perform various modifications to the Internet traffic. Typically these modifications are focused on performing some sort of malicious activity. Ettercap's ability to perform these modifications is of great interest to the development of Apoc.

How Ettercap Works

Once Ettercap has established its MitM position all of the network traffic to an end host will be routed through Ettercap. Ettercap provides a filter engine for which these packets can be passed through and then modified at the user’s discretion. This feature is known as Etterfilter. Etterfilter is a piece of Ettercap that interprets and executes a user provided filter script. This script instructs Etterfilter to perform specific operations to the network traffic. Etterfilter has developed a small language used to express these modifications. This language presents a novel method for performing modifications to network traffic. An simple example of the Etterfilter language is shown below

```plaintext
if(ip.src == '192.168.0.1' && ip.ttl == 23) {
    ip.src = '192.168.0.2';
} else {
    replace("ethercap", "ettracp");
}
```

This program is meant simply to show how the language looks, it does not perform any logical function. The If statement checks each packet to see if the source IP address
is 192.168.0.1 and the TTL is 23. If a packet is received with these characteristics the source IP address is changed to 192.168.0.2. Otherwise the else section is performed. This section contains the replace function. This function searches the packet for the string "ethercap" and replaces it with "ettercap".

The Etterfilter language is an ideal approach to controlling the functionality of Apoc. Apoc has borrowed the Etterfilter language and slightly modified to more accurately fit its needs. Chapter 4 covering the design and implementation of Apoc will further explain how this language was implemented.

Netfilter

Netfilter is a packet manipulation framework built into the Linux kernel. It was first introduced in the 2.4 kernel to provide developers with an API which would allow them to modify packets in the network stack. Currently the most popular use of Netfilter is the iptables firewall. Iptables is a stateful packet filtering firewall for Linux that was built on the Netfilter framework. This section will explain how Netfilter works and how it was useful in Apoc's development.

How Netfilter Works

Netfilter consists of a number of hook functions and network related data structures built into the Linux kernel. The hook functions are typically the most popular feature of Netfilter as they are useful when creating many networking applications. These hooks provide developers with the ability to directly access packets as they traverse the network stack. These hooks are strategically placed into the network stack to provide a significant amount of control over how packets are handled. Each individual protocol is given its own set of hooks, these provide the programmer with control over how packets of that protocol traverse the Linux network stack. The following is a list of common
hook locations.

- **PRE_ROUTING** - When the packet is first read in, before it is processed for routing

- **LOCAL_IN** - If the packet is intended for this host this hook is called before the packet is sent for any processes

- **FORWARD** - If the packet is destined for another host it passes through this hook

- **LOCAL_OUT** - This hook is called for all packets that are created by the local host

- **POST_ROUTING** - This is the final hook placed after routing has occurred

Netfilter's hooks are not utilized in the development of Apoc. However, some of Netfilter's other features have been useful. One main idea in Netfilter which has influenced Apoc is its connection tracking ability. Netfilter implements the necessary data structures to allow iptables to work as a NAT. This means that Netfilter must provide the functionality to translate between the internal and external addresses depending on which interface a packet is sent or received. Netfilter's connection tracking schema is described below.

Netfilter's data structures are used to provide the previously mentioned connection tracking functionality. The core of the connection tracking is the *ip_conntrack* struct. This struct contains connection related information including the status of the connection, a timeout value used to check if the connection is still active, and two pointers to *ip_conntrack_tuple_hash* structs. Each *ip_conntrack_tuple_hash* struct is used to keep track of one end of the connection, it also includes a reverse pointer to the *ip_conntrack* struct. Figure 2.8 shows the Netfilter connection tracking structures.

The *ip_conntrack_tuple_hash* struct contains the connection tracking information in an *ip_conntrack_tuple* struct. The *ip_conntrack_tuple* structs include both the source
and destination IP addresses and ports along with the upper layer protocol used in that connection.

Netfilter uses a hash table to keep track of these connections. The key in the hash function is the `ip_conntrack_tuple` struct. The resulting value from the hash look up is a `ip_conntrack_tuple_hash` which contains a reverse pointer to the main `ip_conntrack` struct. Whenever a new packet is received a `ip_conntrack_tuple` is created with the connection information from that packet, this struct is then hashed and the resulting `ip_conntrack_tuple_hash` is returned. The reverse pointer in this struct can be used to access the `ip_conntrack` struct where the connection information for the opposing side of the connection can be obtained(6).

As every received packet requires a hash lookup, it is imperative that the hash function produces both fast and stable results. Research in hash function performance has yielded the Jenkins hash function as the most appropriate for Netfilter(7). This is significant as Apoc a has need for a hash function with similar qualities.
CHAPTER 3  Practical Uses of Apoc

The goal of Apoc is to implement a framework that will be able to provide any number of possible uses in the future. However, currently there are a number of interesting problems for which Apoc can be applied to. These include creating random values in packets and spoofing addresses. As Apoc's development was inspired for use in the ISEAGE lab, most of its functionality has been focused toward solving current problems in ISEAGE.

Randomization

One of the most prominent features in Apoc is its ability to create random values for various protocol fields. Typically network testbeds lack the pure randomness possessed by the Internet. The ability to produce random fields in packets can create a network environment that appears to be much more chaotic, providing a closer representation to Internet.

The most interesting fields to randomize are IP addresses. Both the source and destination addresses can provide different characteristics when they contain random values. The benefits of this feature is explained in the follow sections.

Random Source IP Addresses

Creating random source IP’s will provide the image that all packets sent to a host are done so from a different source host. This is a very useful means to either anonymize the
true source of a machine or make traffic seem to originate from a number of hosts. This can be especially useful when launching attacks from one host. The ability to anonymize the source of these attacks provides a great deal of usefulness to ISEAGE. Real attacks may originate from anywhere in the world. However, when launching an attack from a network testbed it is much easier to use one host which can make the attack appear to originate from a number of different sources.

A number of cyber attacks require a random source address to be accurately portrayed in an Internet testbed. These attacks are developed in distributed manner, which makes them more difficult to defend against. Examples of distributed attacks include Denial of Service (DoS) attacks and port scans(14).

**Distributed Denial of Service**  DoS attacks focus on decreasing the availability of a service. Often DoS attacks will send a great deal of traffic to a host, overwhelming its capabilities to adequately process the resulting information. If DoS attacks are launched from a single host it is easy to block the source of the attack. However, if the attack can be performed in a distributed manner it is near impossible to defend against.

**Distributed Port Scans**  Port scans are another attack than can benefit from random source addresses. A port scan is probably the most popular method attackers use to gain information on their victim’s machines. Port scans take advantage of the way TCP and UDP handle connections. A port scan sends a packet to different ports on the victim host and uses the response information to see if there is a service running on that host. As port scans are often the first indication an attack is being launched, if they are identified the attacking host could easily be blocked. To combat detection, port scans can be sent in a distributed way so each packet sent to a port seems to originate from a different host.
Cyber Defense Competitions  The ability to send packets with random source IP addresses will be very useful in ISEAGE as it is often used to house Cyber Defense Competitions. In these competitions Red teams are given a range of IP addresses in which to launch attacks from. However, providing a red team with a limited range of IP’s reduces their ability to provide real world attack scenarios as blue teams can easily recognize and block known malicious IP ranges. Apoc will allow the red team to use any desirable IP address or range of addresses making the attacks appear to come from many different areas on the Internet.

Random Destination IP Addresses and TTLs

Although the creation of random destination addresses by itself is not as interesting as random source address, when combined together they provide some very useful functionality, such as creating background traffic. This feature could be coupled with a traffic generation program to generate random background traffic which will provide a more authentic feel to ISEAGE.

Another interesting field to randomize is the IP Time to Live, or TTL field. Since this field specifies the number of times a packet has been routed, it provides a strong indication of how far the source host is away from the destination host. Sending packets with random TTL’s will provide a scenario where the packets arriving to a host will all seem to come from hosts at different distances.

MAC address modification

One problem currently being encountered at ISEAGE is a heavy reliance on hubs for network connectivity. Switches are typically preferred over hubs as they reduce contention in the network, however, they introduce problems in ISEAGE. As ISEAGE aims to provide a realistic Internet environment, it must perform a great deal of traffic
generation. Switches will prevent generated traffic from being seen by many hosts as the switch will only forward the traffic if the packet’s destination MAC address matches a host found on that switch.

Apoc will provide a means to bypass this situation by giving the user the ability to change the MAC addresses of each packet. This MAC address could be changed to either a known host on the switch or to the broadcast MAC address. By sending a packet with a broadcast destination MAC address, each packet will be forwarded through all switches it encounters (Figure 3.1). This helps ISEAGE provide its traffic generation abilities without a heavy reliance on hubs.

![Figure 3.1 Bypassing switches with broadcast MACs](image-url)
CHAPTER 4 Design and Implementation

This section will provide the reader with a greater understanding how Apoc works and the problems that were overcome in its development. It will begin by explaining the basic process of reading, modifying, and writing a packet. It will continue to explain how the Etterfilter language was utilized and how connection tracking is performed. Finally it will conclude with an explanation of major problems encountered during the development process.

Basic structure

The basic structure is implemented with three interfaces. The first two interfaces are used to intercept the traffic between the communicating hosts. These two interfaces are frequently referred to as the filtering interfaces. Typically one interface is connected to ISEAGE, while the other interface will be directly connected to either a single host or a small subnet of hosts through a hub or switch. Apoc's third interface is the control interface. This allows the user to remotely control Apoc without interfering with the network traffic it processes.

Since every packet received on the filtering interface must be forwarded, Apoc implements a very simple packet forwarding scheme. Packets are read in on one interface, some number of modifications are performed to the packet, and the packet is written out the opposing interfaces. As Internet traffic is fully duplexed; Apoc simultaneously reads, modifies, and writes traffic on both interfaces.
As there are two separate interfaces constantly reading and writing packets, Apoc utilizes threads to more efficiently accomplish this process. When Apoc is started both interfaces receive their own thread; this thread is responsible for reading any packets from that interface, modifying them, and writing them out the opposing interface. The use of threads was based upon the ability to write code that more accurately represents the problem.

The thread library used by Apoc is the POSIX Threads (Pthreads) library. Pthreads provides a very straightforward method of spawning and controlling threads. When Apoc begins running each interface receives its own thread; this thread will also receive its own filter script which is must parse and execute. Figure 4.1 shows how Apoc's threaded architecture.

![Figure 4.1 Apoc's threaded structure](image)

**Reading Packets**

Each packet must be read and processed on the filtering interfaces. Apoc needs access to each packet that arrives on these interfaces. This functionality is provided with the Pcap library. The Packet Capture Library or Pcap is described in the man page as "a high level interface to packet capture systems" (8). The Pcap library provides a number of functions that allow a developer to read packets arriving on a specified network interface. Pcap is especially useful when a program is interested in reading
every packet that an interface sees. Pcap places the interface into promiscuous mode so that every packet that arrives on that interface will be read.

When using Pcap the programmer must first specify the interface on which packet capturing is desired, this is performed through the `pcap_open_live` function. This function accepts the name of the interface on which packet capturing will be performed as an argument, it also accepts additional arguments which are not of interest here. Once pcap is initialized with `pcap_open_live`, it returns a packet capture descriptor which is passed to other pcap functions as an argument to specify the device to capture on. Pcap provides a number of functions to read packets, but Apoc only utilizes the `pcap_loop` function. This function is provided the packet capture descriptor received during the `pcap_open_live` function. `pcap_loop` also accepts the name of a callback function as an argument. When a packet is received on the interface Pcap calls the callback function and sends it a buffer containing a entire packet, this function can then be used to process each packet. The `pcap_loop` function is ideal for Apoc as it performs packet capturing in a loop, by continually capturing and processing packets.

After a packet is passed to Apoc by the Pcap library, it will execute the contents of the filter script onto the packet. This begins by first performing some classifications of the packet. The packet and the specific protocols used in that packet are stored in a general struct used for all future packet modification functions.

**Packet Modification/The Etterfilter Language**

The modification of packet in Apoc is based entirely on the scripts provided by the user. The language used by Apoc was heavily based on that used in Etterfilter. This section will explain how this language has been implemented and how it is used to perform modifications to packets.
Control Structures

The modification process was implemented to exactly follow the script specified by the user. This insures that the packet modifications exactly follow the structure intended by the user. Etterfilter implements a number of control structs which represent the basic control structures in the Etterfilter language. There are four basic control structs used to implement this structure; these are block, single_instr, ifblock, and condition. Apoc borrows these structures from Etterfilter, but implements its own end structures used during that actual conditional and instructional statements. The rest of this section will explain the Backus-Naur Form (BNF) for these main structures and describe how they are implemented in Apoc.

**Block** The most abstract component of this language is a block. A block can consist of either an if statement or a single instruction, it also may contains another block. This allows the language to have many different blocks chained together. The Backus-Naur Form (BNF) for a block is shown below.

\[
\text{\langle block \rangle ::= \langle single\_instruction \rangle \langle block \rangle} \\
| \langle if\_statement \rangle \langle block \rangle \\
| \langle if\_else\_statement \rangle \langle block \rangle \\
| \text{NULL}
\]

When a block is found while parsing the language a corresponding block struct is created, this struct contains pointers to the related control structs. This block struct is shown below.

```
struct block {
    struct single_instr *ins;
    struct ifblock *ifb;
    struct block *next;
    int type;
}
```

Figure 4.2 block struct
**Single Instruction**  A *single instruction* is a simple structure that represents some operation to be performed, this will typically signify some modification to a packet. This modification can be either a simple instruction such as changing a value in a protocol header or a more complex change which can be specified through the use of a function. Both of these types of modifications will be described in greater detail later in the document.

\[
\langle \text{single\_instruction} \rangle ::= \langle \text{instruction} \rangle ;
\]

If and if/else statements are used to perform conditional operations. These provide the user with a means to determine if certain condition has occurred when a packet is inspected. This implements control over when a modification will be performed. These conditional statements are contained in an *ifblock* struct.

\[
\langle \text{if\_statement} \rangle ::= \text{if} (\langle \text{conditional\_block} \rangle) \langle \text{block} \rangle
\]

\[
\langle \text{if\_else\_statement} \rangle ::= \text{if} (\langle \text{conditional\_block} \rangle) \langle \text{block} \rangle \text{ else } \langle \text{block} \rangle
\]

Finally a *condition\_block* structure is provided to implement conditional statements. A conditional statement may consist of a number of individual conditions which can be linked together with *and* (\&\&) or *or* (||) operators.

\[
\langle \text{condition\_block} \rangle ::= \langle \text{condition} \rangle
\]

\[
| \langle \text{conditional\_block} \rangle \&\& \langle \text{conditional\_block} \rangle
\]

\[
| \langle \text{conditional\_block} \rangle || \langle \text{conditional\_block} \rangle
\]
Offsets

A key component of the Etterfilter language is an *offset*. Offsets are used to specify fields within protocols. An offset represents a range of bits starting some number of bits from the beginning of the packet. For example, the offset to the source address in an IP packet is the twelfth byte in the IP header and spans through the sixteenth byte. This information can be used to locate the proper data in a packet in which can then be used to perform some operation. Offsets are represented in this language as a pairing of the protocol and fields name in the following format `protocol.field`.

In order for Apoc to transform an offset into usable information it borrows the method used in the Etterfilter language. This method uses a configuration file that is read when the Apoc loads. This file contains the necessary offset information for all the protocols understood by Apoc. Each protocol contains its own section in this file containing the name of the protocol and the level it lives on. For each protocol it contains the name of each field along with the length of that field and the location of the field in the protocol header. Protocols are described with a line like `[ip][3]` stating that the string “ip” maps to the name of a protocol on the network layer. Fields are specified with a line such as `src:4 = 12`, this states that the string “src” is a field in the IP protocol. The size of that field is 4 bytes and the field is found 12 bytes into the IP header. Below is the entire configuration section for the IP protocol.

```
[ip][3]
  hl_ver:1 = 0
  tos:1 = 1
  len:2 = 2
  id:2 = 4
  frags:2 = 6
  ttl:1 = 8
  proto:1 = 9
  csum:2 = 10
```
When Apoc parses the script and finds an offset it fills a struct containing the relevant field information. When performing the operations to a specified offset Apoc looks at this struct and can map the data of the offset, size, and layer to the implied location in the packet.

Offsets are used whenever a field must be specified. They are commonly used in two different methods. The first method where they are used is in a conditional statement. Here some value is compared to an offset to check if the data in the packet is equal to some specified data. The second method is in an assignment statement. Here some value is assigned to an offset. This modifies the corresponding data in the packet to contain the data specified in the assignment statement.

Functions

The Etterfilter language also lets the user perform special modifications with the use of functions. A function produces a mechanism in which user can specify more abstract concepts. Currently function support is limited to only supporting the randomize and drop functions, these are described below.

- randomize – This function tells Apoc to create a random value for some field. The first argument to this function is the offset of the field which will be changed. The second and third let the programmer specify the lower and upper limit of the random value. The random values are received by reading from the /dev/urandom device on the system.

- drop - This function simply drops a packet.
Scanning and Parsing the language

Apoc must be able to interpret the information provided in the filter script. Ettercap already has performed a great deal of the preliminary work necessary to interpret the Etterfilter language, including the scanning and parsing. Apoc is able to modify these techniques for its own needs. The process of scanning and parsing this language is described below.

**Flex**

Flex is an open source tool used to generate a lexical analyzer. A lexical analyzer or scanner is a tool that performs pattern matching on some input. Flex is provided a configuration file explaining the various distinct patterns that occur in a language. These distinct patterns are often known as tokens. The lexical analyzer will then search through an input and return each individual token it reads. The tokens can be either directly specified or described through regular expressions. Tokens that are directly specified are concrete terms such as `if`, `else`, `;` etc. Regular expressions are used to match tokens that are not concrete and must be matched through their patterns. For example, in a typical programming language variables and function names are matched through this method. Here regular expressions are extensively used to match different types of data formats. An example regular expression from the Etterfilter language is `\([0-9]{1,3}\).[0-9]{1,3}\).[0-9]{1,3}\).\([0-9]{1,3}\)` This expression matches an IPv4 address. Whenever the lexical analyzer sees an input which matches this pattern it will return an IPv4 token.

Flex works by first reading a user generated input file which describes each individual token in the language. After reading this file Flex generates the lexical analyzer in C. Once the lexical analyzer has been created it can be called by the `yylex()` function. Each time this function is called it will return the next token from the language. Typically this function is called from the a parser such as Bison which is explained in the next section.
Bison  Bison is an open source parser generator. Once the grammar for the language is specified Bison will generate a C program that will parse that grammar. To create the parser generator the user must specify the grammar in a configuration file. Apoc extends the configuration file from the Etterfilter language to generate the parser. The Bison parser is dependent on Flex for obtaining tokens. Bison receives tokens from calls to Flex’s `yylex()` function, as Bison receives tokens from Flex it matches the received tokens to a set of rules specified in the grammar.

Bison creates parsers for context free grammars. A context free grammar consists to two types of statement, terminals and non-terminals. Terminal statements are the atomic units in the language. These terminal statements are the lexical tokens returned by the Flex scanner. Non terminal statements are not atomic, they can consist of combinations of non-terminals and terminals, these represent the way a language constructs groups of tokens to create meaningful statements. Context free grammar are often displayed in BNF syntax. Here all of the non-terminals are shown on the left side of a rule and their definitions are displayed on the right side. Examples of the BNF syntax were shown in the Control Structure section.

Modifications to the language

The language used by Apoc is almost identical to that produced for Etterfilter. However, there are a few areas where the language has been modified to meet Apoc’s needs.

Apoc has added extra tokens to represent desired data formats. The first token added is a MAC address token. This lets the user specify MAC addresses in their typical colon delimited format. The second token added is a CIDR address token. This provides the user with a means to match entire ranges of IP addresses.

Another area where the Etterfilter language falls short of Apoc’s needs is that it does not perform parsing of function arguments. This is a desirable feature as many function
arguments are tokens in this language. By parsing these values Apoc will perform any necessary data conversion to that token. This greatly reduces the overhead in function writing as the developer does not perform any data conversion techniques.

Writing Packets

Once a packet is read in and any relevant modifications have been performed the packet is ready to be written to the interface. This is accomplished with Libnet library. Libnet has a more programmer friendly API than the traditional Unix sockets and provides functions useful in writing packets to an interface. All packet writing in Apoc is performed thought the Libnet library. In order to create a packet with Libnet the user must first initialize a Libnet context, this context is the basic structure used control the packet creation process. A Libnet context is created with the `libnet_init` function, this function also accepts the type of injection to be performed. Apoc uses the injection type `LIBNET_LINK_ADV`, which provides Apoc the ability to send packets on the link layer.

Apoc uses Libnet in a method that varies slightly from its traditional use. Libnet is generally used to create packets from scratch. However, since Apoc has already has a fully developed packet it has no need to recreate the packet. Apoc only needs a method to write the already developed packet to the network. If the packet has been modified the proper protocol checksums must be recomputed. The `libnet_do_checksum` function is used to recompute these checksums.

Once the checksum is created the `libnet_adv_write_link` function is used to write the packet. A pointer to the packet and the length of the packet is passed to this function.
Connections Tracking

In order for Apoc to accurately perform its duties it needs to be a very stateful system. This is largely due to the connection oriented paradigm present in many networking protocols. As a result changing a certain field in a packet may result in what appears to be an entirely different connection to the machine receiving that packet. A good example of this are IP addresses and TCP/UDP ports. A change to an IP address tells the packet it is destined for an entirely different machine, while a change to a port number will result in a packet being directed towards a different application on the host. If any of these sensitive fields are changed it requires that all future packets in this connection also perform these changes.

Connection tracking was discussed earlier in the section covering Netfilter. Apoc has very similar needs to Netfilter for connection tracking. Netfilter's connection tracking schema was a strong influence in Apoc’s development. While Apoc uses Netfilter’s connection tracking for its basic design it simplifies the idea by limiting the number of structs used and eliminating some unneeded fields.

Apoc uses two main structs for connection tracking, `conntrack` and `conn_end`. These are displayed in figure 4.4. The `conntrack` struct contains pointers to two different `conn_end` structs. The first `conn_end` struct contains the connection information which matches how packets will be seen on the first interface. The second `conn_end` struct contains the connection information as seen on the second interface.

![Figure 4.4 Apoc connection tracking structures](image-url)
Connection Tracking Algorithm

This section will describe the how the connection tracking is actually performed. First a conn_end struct is created for every new coming connection. This struct is then hashed to retrieve the corresponding conntrack struct. If no hash entry is available it can be assumed that this is a new connection, however, if an entry is found for that hash table it can be concluded this packet is part of an already established connection. These two situations will be described in the following two paragraphs.

If the packet is part of a new connection a conntrack struct will be created for that packet. Then any necessary modifications will be performed to this packet. After these modifications are performed the reverse conn_end struct will be created. If any connection related modifications were performed this conn_end will represent the changes to the connection. This conn_end struct will then be inserted into the hash table with the location of the conntrack struct as the hash value.

If the initial hash lookup results in a non null value it is assumed that the packet is part of an existing connection. The return value of the hash lookup will be the address of the corresponding conntrack struct. After this struct is obtained the necessary modifications will be applied to the packet. After the modifications are performed the reverse conn_end is obtained from the conntrack struct and that connection information is applied to the packet.

This process has the negative effect of allowing only static connections throughout the life of the Apoc session. Once a connection is created it will remain intact indefinitely. There is no means to further modify a connection after it has been established.

Hash Table

The hash function used in Apoc is the same Jenkins hash function used in Netfilter. This function provides an ideal hashing algorithm for Apoc's needs. Since Apoc's
connection tracking hash is very similar to Netfilter's, it can be safely assumed that this hash function will provide Apoc with optimum performance. Apoc does not use the code directly from Netfilter. Instead, Apoc uses a slightly different implementation of the Jenkins hash function. Although this code implements the same algorithm, it provides a slightly different interface to the functions. This implementation has been borrowed from the XML-Lit project from Sourceforge (9).

Problems

A number of interesting problems were encountered in the development of Apoc. Two interesting problems referred to as packet reiteration and ambiguous connections. This section will describe these problems in detail and explain how they were overcome in Apoc.

Packet Reiteration

Apoc has very similar characteristics to a network gateway as they are both used to forward traffic between two hosts. However, Apoc differs in that packets are never addressed directly to it. Apoc reads packets addressed to other machines and forwards them, meaning that Apoc must read packets in promiscuous mode. Apoc's reliance on interfaces in promiscuous mode presents an interesting problem, which is referred to as packet reiteration.

Packet reiteration occurs when Apoc writes a packet to an interface which is reading and forwarding every packet it sees. Every time a packet is written to an interface, it will be immediately read by the opposing interface. Since this interface also reads and forwarding packets, every packet will continually be passed through the system. Figure 4.5 below shows the problem of packet reiteration as a packet will travel in a cycle throughout Apoc indefinitely.
To address this problem Apoc must be able to identify packets which have already been processed. Once these packets are identified they need to be dropped. Apoc handles this situation by remembering recently written packets. Each time a packet is read on an interface it can be compared to the packets recently written to that interface. If there is a match it can be assumed that the packet was already processed by Apoc and can be safely dropped.

To elaborate on this problem it is important to describe the workings of the network stack on a typical operating system. Modern operating systems such as FreeBSD and Linux place packets into a queue directly after they are read from the interface. When a packet is read it is added to the queue where it waits to be processed. When the operating system is ready to process the packet it will take the packet from queue. This means there may be a number of packets waiting to be processed by Apoc at any one moment.

Apoc implements a queue of packets which have been recently written. This queue is used to mirror the queue in the operating system. Each packet written by Apoc is appended to this queue, while each new packet read is compared with the first packet in this queue. A match signifies that a packet has been previously processed and needs to be dropped from further processing. Figure 4.6 shows how an incoming packet is compared with recently written packets.
Figure 4.6 Avoiding packet reiteration

The queue of recently written packets is implemented with a linked list. The list's ends are moderated with a *written_pkts* struct, this struct maintains a pointer to the first and last packet in the queue. Packets are represented with a *w_pkt* struct. This struct simply contains a pointer to a packet, the length of that packet, and a pointer to the next *w_pkt* in the queue. Each interface maintains its own queue for the packets written to that particular interface. The head packet on the queue is compared to each packet read on that interface allowing each interface to efficiently identify and eliminate any repeat processing of the packet.

**Ambiguous Connections**

Apoc's connection tracking ability was established previously in this section. Although this is an interesting feature it can hinder the natural packet flow through Apoc. It is important to note that connection tracking is only possible if each connection end is entirely unique. An ambiguous connection will be created if a packet cannot be properly forwarded due to insufficient information on that connection.

Modifications to source IP addresses will indirectly lead to a problem with ambiguous connections. Here we will refer to an IP that has been modified by Apoc as an
unnatural IP. The problem is that other protocols such as ARP will use this unnatural IP. Since ARP does not utilize a port number it cannot properly utilize the connection tracking functionality. The next paragraph will explain how ARP packets can create an ambiguous connection.

If a host wants to respond to a packet received from an unnatural IP it will ARP for that MAC address. This ARP packet must be forwarded through Apoc, which needs to change the unnatural IP to the original IP address. The ARP reply will then be forwarded through Apoc changing the original IP address back to the unnatural IP. Since ARP does not contain a port number it cannot access the necessary connection information. If there is more than one connection using the original address, Apoc will not be able to decide with connection that ARP packet belongs to. Hence, the packet cannot be forwarded. Here an ambiguous connection has been developed.

To handle this situation Apoc implements another hash table. This is used to map unnatural IP addresses to the MAC addresses which they originated from. The IP from every ARP request is then looked up in this table. If a match is found Apoc will drop the ARP request and continue to send its own ARP reply with the MAC address it received from the table. While Apoc naturally seeks to be a transparent entity, this is a situation where it is required to usurp the natural flow of the network traffic.
CHAPTER 5 Results

This section intends to show the results of Apoc. It will begin by showing how Apoc is used. It will then show the current state of Apoc’s functionality in its ability to handle problems in which it was developed to solve. While Apoc’s utility extends well beyond these core problems, most of these features have not been sufficiently tested to provide any feedback on their effectiveness.

Usage

Currently Apoc allows the user to provide two options when run. These options are specified by the -o and -l flags. The -o flag is followed by the filename of the filter script to be executed by first interface on that system, while the -l flag accepts the filename for the second interface’s filter script. Examples of Apoc being run are shown below.

```
./apoc
./apoc -l script2
./apoc -o script1 -l script2
```

The first example executes Apoc without any filter scripts, it will perform basic packet forwarding. The second example show Apoc being run with just a filter script for the second interface, here the first interface will perform basic forwarding. The third example shows Apoc with a filter script for both interfaces.

The filter scripts can provide a wide range of functionality. Some example scripts are shown below.
If (tcp.dst == 80){
    ip.dst = '192.168.1.100';
}

This first script will tell Apoc to work as a Destination NAT. Here all packet that are sent to a web server can be redirected to a specified server at 192.168.1.100.

random[ip.src, '1.1.1.1', '255.255.255.255'];
random[ip.dst, '1.1.1.1', '255.255.255.255'];
random[ip.ttl, 5, 127];

This script modifies packets to have completely random source and destination IP addresses along with random TTLs. This might be useful in random traffic generation.

if(ip.dst == '198.41.0.4' || ip.dst == '128.8.10.90' || ip.dst == '192.33.4.12'){
    ip.dst = '10.10.10.2';
}

This script is used to reroute packets destined for a particular host to another host. In this example packets sent to root DNS servers are routed to another server.

if(ip.csum == 0x1234){
    ip.csum = 0x1235;
}
else{
    ip.csum = 0x1234;
}

This script can be used to send packets with bad IP checksums. This could used to test how a machine handles malformed packets.

if(ip.dst == '192.168.0.0/16'){
    eth.dst = 'ff:ff:ff:ff:ff:ff';
}
This script changes the MAC address to the broadcast address if a packet is destined for the 192.168.0.0/16 address range. This could be useful to convince a switch to forward the packets.

**Network Performance**

Network performance is a very important feature of Apoc. For Apoc to be a useful tool it is necessary that it does not introduce any significant performance degradation into the network. In order to accurately measure Apoc’s performance two different tests were performed.

The first test is was performed with Iperf, which is tool designed specifically to test the bandwidth of a connection. Here one host is configured as a server and another as a client. Then Iperf client sends 8-KB data packets to the server for 10 seconds. At the end of this period Iperf reports how much data was received at the server. This test was performed four times with a straight connection between two hosts and then four times having all packets passing through Apoc. The straight through connection produced an average of 55.5 Mbits/sec while the Apoc test managed only 3.75 Mbits/sec.

The previous testing method was fairly one dimensional. It only created one TCP connection which all data was passed through. This a poor test for two reasons. First, it does not test Apoc’s ability to handle a large number of connections, and second, it does not adequately represent typical Internet traffic. Due to these reasons another method has been created to more thoroughly test Apoc’s ability to handle more abstract network traffic. This test uses Nmap to scan a host. Nmap makes for an excellent test tool because it will create thousands of connections placing Apoc under a high load. Nmap also reports how long the scan took, which can be used to determine the performance of Apoc.

Nmap scans were performed first on a direct connection, then through Apoc with
any modifications, finally four scans were run with Apoc creating random source IP addresses with the following script.

```
random[ip.src, '10.1.1.1', '10.10.10.255'];
```

The nmap scans were performed with the `-sT` option to create full TCP connections, each test scanned 1667 ports. The direct scans averaged 10.885 seconds per scan while scans through Apoc averaged 12.658. Performing random source IP address creation significantly increased the length of the scan to 24.554 seconds.

From these results it is apparent that there are serious performance problems in the operation of Apoc. While the nmap test performed appropriately the results of the Iperf test were unacceptable. The source of the bottleneck is currently unknown. Initial testing has shown that neither the connection tracking or the packet reiteration modules are responsible for these results. A current hypothesis is that the problem may be the direct result of the threaded nature of the code. This needs to be further researched before Apoc can be considered a reliable tool.

It should also be noted that there was a large discrepancy in the machines used to test Apoc. Apoc was run on a 1Ghz Pentium III with 384MB of RAM, running Ubuntu 5.04. One end node was represented by a 1.8 Ghz Pentium Centrino with 512MB of RAM running Debian Linux. The opposing host is a 233 Mhz Pentium II with 32 MB of RAM running FreeBSD 5.4.

**Testing**

The testing of Apoc has been performed from a number of angles. Currently testing on Apoc has been primarily focused on the correct implementation of the language features. A number of scripts have been developed to attempt to fully test different aspects of the language. These tests were performed by replaying large amount of
network traffic with approximately thirty thousand packets at a slow speed as to not incur any dropped packets.

The first test was developed to check Apoc's ability to handle nested if statements and various operators such as $\geq$, $<$, and $>$. This test script is shown in Appendix A, Testing Nested If Statements. The script checks the value of the IP checksum, which it then used to attempt to weight the modification of source IP addresses by using a number of if statements. The result of this test shows that every modified source IP is found in the resulting network traffic. This result of this test provides evidence that both nested if statements and operators are implemented correctly. More information concerning this test is also found in Appendix A.

After Apoc's ability to accurately modify a particular field was derived, further testing was performed to provide feedback on the implementation of the conditional statements. This test was developed to analyze Apoc's ability to handle multiple conditional statements. An Apoc script was developed with two separate if statements, the first if statement contained three conditions grouped together with the and operator. The second if statement contained three conditions grouped together with the or operator. The same data from the previous test was then replayed through Apoc to check when the conditions were recognized. If a packet matching a condition was found, it was marked by modifying the Ethernet header to be either all 0's or 1's. After the test was run the original packets were loaded into Ethereal and the packets were passed through the same conditional statements on Ethereal. This provides that actual number of packets which met the specified statements. This test had negative results. Certain packets were marked by Apoc, without meeting the required conditions. This signifies that Apoc currently does not handle conditional statements correctly. Some results from this test can be viewed in Appendix A, Testing Conditional Statements.

The final group of tests focused on the correct implementation of the offsets supported by the language. Currently there are six protocols supported in Apoc; Ethernet, ARP,
IP, ICMP, TCP, and UDP. A separate test has been developed for each protocol. The tests were intended to assure that the packet values modified by Apoc directly match the bits modified in the packets. Each modifiable field was tested on each protocol. After each test the packets were reviewed to assure that they conformed to the fields specified in the Apoc script. All results of the this test were favorable. Some results from these tests can be seen in Appendix A, Testing Proper Offsets.

Future Testing

Although replaying network traffic provides a significant amount of information on Apoc’s functionality, it falls to test certain features. Since all replayed traffic originates from one host, Apoc is unable to properly perform connection tracking. Hence, connection tracking cannot be properly tested with replayed network traffic. To accurately test Apoc’s connection tracking ability it must have access to live network traffic. Also, tests with live network traffic provides the only means to assure that Apoc will operate effectively in a real environment. Thorough tests with live network traffic have not yet been performed. The following section will provide the scenario for which future testing of Apoc should undergo.

First their should be multiple hosts communication through Apoc. While it is not feasible to setup a large number of hosts for a test, a more moderate number is proposed. For this test eight hosts will be used, with four on each side of Apoc. These can be configured with multiple IP addresses to emulate more hosts. Various servers shall be set up on these hosts, which will be accessed to create the network traffic.

Once the test machines are configured, the test traffic should be carefully selected. This traffic must be chosen to accurately portray all types of network traffic. This traffic should contain large amounts of IP, ARP, TCP, UDP, and ICMP packets to test all of Apoc’s supported protocols. However, testing should also include protocols not supported by Apoc, such as IPv6 or IPsec. This will test Apoc’s ability to handle
unsupported protocols. Traffic shall consist of large file transmissions such as FTP session as well as smaller transmissions such as a port scans. Port scans are particularly useful in testing the connection tracking ability. It should also include other common traffic such as HTTP, SSH, and DNS.

Test traffic should also contain a significant amount of malicious traffic, as Apoc will often be used to forward attacks. This will test Apoc’s ability to handle traffic which deviates from typical Internet traffic. One major source of malicious traffic should originate from various vulnerability scanning tools as Apoc will often be used to forward traffic from these scans during Cyber Defense Competitions. This traffic shall also include any traffic which may be used to avoid IDS systems, as these attacks contain traffic with very unusual fingerprints, which may disrupt the flow of traffic through Apoc.

Once the traffic has been sent there must be an efficient method to determine if all traffic was accurately received. The most obvious method is derived from the accessibility of the network services. If all services work correctly there will be evidence that all data is being properly forwarded. More evidence of this can be found if by saving pcap files of all network traffic. By interpreting these files in Ethereal individual connections can be traced. Comparing both sides of a connection in this manner will provide information if the traffic was properly forwarded or if any packet loss was incurred.

The previously explained testing methodologies will provide sufficient evidence of Apoc’s functionality. Current tests on Apoc show that there are some areas which are still problematic. However, future tests are need to provide adequate feedback concerning Apoc’s functionality.
CHAPTER 6 Summary

The goal of Apoc was to provide the user with absolute control over the traffic passed through it. Currently Apoc provides many methods through which this traffic can be modified. The basic problems which spawned Apoc's development such as changing MAC and IP addresses can be easily performed. Apoc's functionality extends well beyond these features to provide control of any information found in popular Internet protocols.

In the Results section it was established that many features have been tested with favorable results. Apoc performs many packet modifications as desired and is able to maintain the proper state by performing connection tracking. However, it was also established that Apoc suffers from some significant performance problems and also has small problems with language interpretation. This problem may hinder Apoc's ability to perform its required tasks.

There are numerous areas where Apoc will benefit from extended work. However, due to time constraint work on Apoc has been limited to the scope covered in this paper. These areas extend from additional features which will extend Apoc's packet modification abilities to more rigorous testing which will ensure Apoc's survivability. Areas that will particularly benefit from extended work are covered in the next section.
CHAPTER 7 ~ Future Work

One key idea in the development of Apoc was flexibility. Since Apoc will have complete access to the traffic intended to a group of hosts it has the potential for a number of more advanced features. Apoc has been designed in a method that will provide future users with the ability add functions without an overwhelming amount of overhead.

Additional Protocols Support

While Apoc currently can handle most popular protocols it still lacks full support in this area. Two popular protocols that remain unsupported in Apoc are 802.11 and IPv6. Although these protocols are fairly popular they are not as commonly used as Ethernet and IPv4. Both IPv6 and 802.11 are rather complex and will be difficult to implement.

Performance

Apoc’s performance results are far from ideal. Although performance was taken into account during the design and implementation, a detailed review of the code will most likely yield areas which could be streamlined. As Apoc will potentially be used on networks with high data rates it is important that its efficiency is maximized.
Packet Data Modification

One useful feature which Apoc lacks is the ability to perform modifications to a packet’s application data. Support for this could be provided with offsets as was done with lower layer protocols. Also general search and replace functions should be implemented which could be used to perform these tasks.

Testing

Testing for Apoc was performed in a very limited environment. Much more testing should be performed to ensure that Apoc is able to properly handle large amounts of traffic from numerous hosts. This testing should be performed in an environment with a large number of hosts and a significant amount of network traffic.

Currently Apoc has only been tested on Debian Linux based systems, however it would ideally run on any Unix system. Many machines in the ISEAGE lab utilize the FreeBSD operating system, therefore, most future testing of Apoc should be performed on this OS.
APPENDIX A  Test Results

This Appendix contains previously mentioned test results. The goal of the following tests was to assess the implementation of the language.

Testing Nested If Statements

The first test was introduced to analyze Apoc's ability to handle nested if statements. It also tests some of Apoc's comparison operators such as > and <. This script attempts to modify source IP addresses based on a weighted distribution by the IP checksum. The script and results are shown in figure A.1.

The results of this test show that all if conditions in this statement were reached as
all of the resulting IP addresses were found in various packets. It should be noted that the functionality of this script cannot be determined by the distribution of the resulting source IPs. Connection tracking obviates the randomness of the checksum field as once a packet is part of a connection Apoc disallow any modification to the IP address.

Testing Conditional Statements

The following test was developed to analyze the conditional statements in Apoc. Apoc allows the user to string together conditional statements with and or or operators. This test contained three conditional statements grouped with the and operator and three conditional statements grouped together with the or operator. This results of this test were poor.

Apoc Script

```apoc
if (ip.src == '192.168.21.3' &&
ip.dst == '192.168.25.1' &&
icmp.type == 81) {
  eth.dst = '00:00:00:00:00:00';
  eth.src = '00:00:00:00:00:00';
  eth.proto = 0;
}
if (ip.csum == 51225 ||
  ip.csum == 11013 ||
  ip.csum == 15234) {
  eth.dst = '11:11:11:11:11:11';
  eth.src = '11:11:11:11:11:11';
  eth.proto = 0x1111;
}
```

Resulting packets

<table>
<thead>
<tr>
<th>Packet</th>
<th>Source IP</th>
<th>Destination IP</th>
<th>ICMP Type</th>
<th>Flags</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>192.168.21.3</td>
<td>192.168.25.1</td>
<td>81</td>
<td>0</td>
<td>51225</td>
</tr>
<tr>
<td>5</td>
<td>192.168.21.3</td>
<td>192.168.25.1</td>
<td>81</td>
<td>0</td>
<td>11013</td>
</tr>
<tr>
<td>7</td>
<td>192.168.21.3</td>
<td>192.168.25.1</td>
<td>81</td>
<td>0</td>
<td>15234</td>
</tr>
<tr>
<td>9</td>
<td>192.168.21.3</td>
<td>192.168.25.1</td>
<td>81</td>
<td>0</td>
<td>51225</td>
</tr>
<tr>
<td>11</td>
<td>192.168.21.3</td>
<td>192.168.25.1</td>
<td>81</td>
<td>0</td>
<td>11013</td>
</tr>
<tr>
<td>13</td>
<td>192.168.21.3</td>
<td>192.168.25.1</td>
<td>81</td>
<td>0</td>
<td>15234</td>
</tr>
</tbody>
</table>

Figure A.2 Inaccurate conditional statements

The first two packets are a result of a match from the first if statement. The first packet matched correctly. However, the second packet is an error. The byte highlighted in red should be 0x08 as is specified by the icmp.type line in the script, however, here it is 0x49. This signifies that an incorrect packet as passed through the conditional statement.
The third and forth packets are a result of a match from the second if statement. The red highlighting shows that these packets are not IP packets, but rather ARP packets. The script should only match IP packets, meaning this packets were incorrectly passed through the conditional statement.

**Testing Proper Offsets**

The results in this section were obtained from tests focused on determining if each offset in the language accurately maps to the proper area in a packet. When performing these tests Apoc seemed to accurately perform the necessary modification in every scenario. Figure A.3 through A.8 show the results from various protocols.

<table>
<thead>
<tr>
<th>Apoc Script</th>
<th>Resulting packets</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eth.dst = '00:00:00:00:00:00';</code></td>
<td><code>0.012094 11:11:11:11:11:11 -&gt; 00:00:00:00:00:00 0x2222 Ethernet II</code></td>
</tr>
<tr>
<td><code>eth.src = '11:11:11:11:11:11';</code></td>
<td><code>0000 00 00 00 00 00 00 11 11 11 11 11 11 22 22 45 00 E...E.</code></td>
</tr>
<tr>
<td><code>eth.proto = 8?38;</code></td>
<td><code>0010 00 34 99 3b 40 00 3f 06 ef 2c 00 a8 8b 02 06 44,403.</code></td>
</tr>
<tr>
<td><code>eth.proto = 8?38;</code></td>
<td><code>eth.proto = 8?38;</code></td>
</tr>
<tr>
<td><code>eth.proto = 8?38;</code></td>
<td><code>eth.proto = 8?38;</code></td>
</tr>
</tbody>
</table>

Figure A.3 Modifying Ethernet fields
Figure A.4 Modifying Arp fields

Figure A.5 Modifying IP fields
Apoc Script
icmp.type = 0x00;
icmp.code = 0x11;
icmp.csum = 0x2222;
icmp.id = 0x3333;
icmp.seq = 0x4444;

Apoc Script
tcp.src = 0x0000;
tcp.dst = 0x1111;
tcp.ack = 0x33333333;
tcp.offset = 0x44;
tcp.flags = 0x55;
tcp.urg = 0x8888;

Figure A.6 Modifying ICMP fields
Figure A.7 Modifying TCP fields
Apoc Script
udp.src = 0x0000;
udp.dst = 0x1111;
udp.csum = 0x3333;

Resulting packets
1 0.000000 192.168.23.4 -> 129.186.140.200 UDP Source port: 0 Destination port: 4369
udp.src = 0x0000; udp.dst = 0x1111; udp.len = 0x0400; udp.csum = 0x3333;
Resulting packets
1 0.000000 192.168.23.4 -> 129.186.140.200 UDP Source port: 0 Destination port: 4369
udp.src = 0x0000; udp.dst = 0x1111; udp.len = 0x0400; udp.csum = 0x3333;

Figure A.8 Modifying UDP fields
BIBLIOGRAPHY


ACKNOWLEDGEMENTS

This work would not have been possible without the support of a number of individuals. I would like to recognize and thank the following people. Dr. Doug Jacobson, for the opportunity to work with the ISEAGE project and constant support of my work. Both Dr. Cliff Bergman and Dr. Tom Daniels for their participation on my thesis committee and valuable insights. Mom, Dad, Emily, and Sam for putting up with me in my struggles and for the endless encouragement. My friends for never giving me too much grief for constantly staying home to work on my thesis. Finally, Matt for the name Apoc.

Thanks.