NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN IMPROVED TARPIT FOR NETWORK DECEPTION

by

Leslie Shing

March 2016

Thesis Co-Advisors:	Robert Beverly
	Justin P. Rohrer
Second Reader:	Mark Gondree

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	03-25-2016	Master's Thesis

4. TITLE AND SUBTITLE

AN IMPROVED TARPIT FOR NETWORK DECEPTION

5. FUNDING NUMBERS

RCJ66
H98230221650

6. AUTHOR(S)

Leslie Shing

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

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

LTS
8080 Greenmead Dr, College Park, MD, 20740

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Networks are constantly bombarded with malicious or suspicious network traffic by attackers attempting to execute their attack operations. One of the most prevalent types of traffic observed on the network is scanning traffic from reconnaissance efforts. This thesis investigates the use of network tarpits to slow automated scanning or confuse human adversaries. We identify distinguishing tarpit signatures and shortcomings of existing tarpit applications as uncovered by *Degreaser* (a tarpit scanner), and implement improved features into a new tarpit application called *Greasy*. We conduct several experiments using a select set of metrics to measure the impact of implementing new tarpitting capabilities and other improvements in *Greasy*, particularly *Greasy's* ability to deceive *Degreaser*, degree of stickiness compared to *LaBrea*, and potential processing overhead as observed by packet latency. Our experimental results show that we effectively mitigate the two tarpit signatures used by *Degreaser's* tarpit identification heuristics. And although *Greasy* may not hold the stickiest connections, compared to *LaBrea* in persist mode, it successfully improves its tarpitting capabilities, while still evading detection. More importantly, the above results are obtained by deploying *Greasy* on an Internet-facing /24 subnet; this allows us to measure *Greasy's* ability to interact with real-world network traffic. Furthermore, *Greasy* offers a modularized extensible tarpit platform for future tarpit development.

14. SUBJECT TERMS

network deception, improved tarpit, Greasy, Degreaser, LaBrea

15. NUMBER OF PAGES 99

16. PRICE CODE

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

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN IMPROVED TARPIT FOR NETWORK DECEPTION

Leslie Shing, Civilian
B.S., University of California, San Diego, 2011

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2016

Author: Leslie Shing

Approved by: Robert Beverly
Thesis Co-Advisor

Justin P. Rohrer
Thesis Co-Advisor

Mark Gondree
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Networks are constantly bombarded with malicious or suspicious network traffic by attackers attempting to execute their attack operations. One of the most prevalent types of traffic observed on the network is scanning traffic from reconnaissance efforts. This thesis investigates the use of network tarpits to slow automated scanning or confuse human adversaries. We identify distinguishing tarpit signatures and shortcomings of existing tarpit applications as uncovered by *Degreaser* (a tarpit scanner), and implement improved features into a new tarpit application called *Greasy*. We conduct several experiments using a select set of metrics to measure the impact of implementing new tarpitting capabilities and other improvements in *Greasy*, particularly *Greasy's* ability to deceive *Degreaser*, degree of stickiness compared to *LaBrea*, and potential processing overhead as observed by packet latency. Our experimental results show that we effectively mitigate the two tarpit signatures used by *Degreaser's* tarpit identification heuristics. And although *Greasy* may not hold the stickiest connections, compared to *LaBrea* in persist mode, it successfully improves its tarpitting capabilities, while still evading detection. More importantly, the above results are obtained by deploying *Greasy* on an Internet-facing /24 subnet; this allows us to measure *Greasy's* ability to interact with real-world network traffic. Furthermore, *Greasy* offers a modularized extensible tarpit platform for future tarpit development.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Research Questions . . . 3
1.3 Summary of Contributions and Findings . . . 3
1.4 Thesis Structure . . . 3

2 Background **5**
2.1 Cyber Deception . . . 5
2.2 Honeypots . . . 6
2.3 Network Tarpits . . . 8
2.4 Related Work . . . 13

3 Requirements and Design **17**
3.1 Requirements . . . 17
3.2 Improvements . . . 19
3.3 Design . . . 25
3.4 Implementation . . . 36

4 Experiments and Results **37**
4.1 Metrics . . . 37
4.2 Experimental Setup . . . 41
4.3 Results and Analysis . . . 45

5 Conclusion and Future Work **65**
5.1 Conclusion . . . 65
5.2 Future Work . . . 66

List of References **71**

Initial Distribution List **79**

List of Figures

Figure 3.1 UML Class Diagram of *Greasy*. 26

Figure 3.2 Multi-Threaded Design. 35

Figure 4.1 Experimental Setup for *Greasy* and *LaBrea*/24 Subnets on the /17 Network Telescope 42

Figure 4.2 Number of Packets Received per /24 Subnet of the /17 Subnet on 10/27/15. 43

Figure 4.3 Boxplot of Packets Sent per Flow (*Greasy* versus *LaBrea* Non-Persist) 48

Figure 4.4 CDF of Packets Sent per Flow (*Greasy* versus *LaBrea* Non-Persist) 48

Figure 4.5 Boxplot of Packets Received per Flow (*Greasy* versus *LaBrea* Non-Persist) 50

Figure 4.6 CDF of Packets Received per Flow (*Greasy* versus *LaBrea* Non-Persist) 50

Figure 4.7 Boxplot of Packets Sent per Flow (*Greasy* versus *LaBrea* Persist) 51

Figure 4.8 CDF of Packets Sent per Flow (*Greasy* versus *LaBrea* Persist) . . 52

Figure 4.9 Boxplot of Packets Received per Flow (*Greasy* versus *LaBrea* Persist) 53

Figure 4.10 CDF of Packets Received per Flow (*Greasy* versus *LaBrea* Persist) 53

Figure 4.11 Boxplot of Duration per Connection (*Greasy* versus *LaBrea* Non-Persist) 55

Figure 4.12 CDF of Duration per Connection (*Greasy* versus *LaBrea* Non-Persist) 55

Figure 4.13 Boxplot of Duration per Connection (*Greasy* versus *LaBrea* Persist) 57

Figure 4.14 CDF of Duration per Connection (*Greasy* versus *LaBrea* Persist) 57

Figure 4.15 Boxplot of Duration per Connection (Same /24 Subnet) 58

Figure 4.16 CDF of Duration Per Connection (Same /24 Subnet) 58
Figure 4.17 Boxplot of Packets Sent per Flow (Same /24 Subnet) 59
Figure 4.18 CDF of Packets Sent per Flow (Same /24 Subnet) 59
Figure 4.19 Boxplot of Packets Received per Flow (Same /24 Subnet) 60
Figure 4.20 CDF of Packets Received per Flow (Same /24 Subnet) 60
Figure 4.21 PDF of RTT Measurements from Run 1 of Latency Experiment Using Residential Vantage Point. 62
Figure 4.22 PDF of RTT Measurements from Run 2 of Latency Experiment Using Residential Vantage Point. 62
Figure 4.23 PDF of RTT Measurements from Run 3 of Latency Experiment Using Residential Vantage Point. 62
Figure 4.24 PDF of RTT Measurements from Run 1 of Latency Experiment Using Naval Postgraduate School (NPS) Vantage Point. 63
Figure 4.25 PDF of RTT Measurements from Run 2 of Latency Experiment Using NPS Vantage Point. 63
Figure 4.26 PDF of RTT Measurements from Run 3 of Latency Experiment Using NPS Vantage Point. 63
Figure 4.27 PDF of RTT Measurements from Run 1 of Latency Experiment Using Local Subnet Vantage Point. 64
Figure 4.28 PDF of RTT Measurements from Run 2 of Latency Experiment Using Local Subnet Vantage Point. 64
Figure 4.29 PDF of RTT Measurements from Run 3 of Latency Experiment Using Local Subnet Vantage Point. 64

List of Tables

Table 3.1 The List of OUIs and Corresponding Index into *Greasy*'s Array of OUI Used to Generate the Upper 3 Octets of a Tarpit Host's MAC Address 21

Table 4.1 Number of Real Hosts versus Tarpit Hosts Identified by *Degreaser* 46

Table 4.2 Average RTT Measurements from Run 1 62

Table 4.3 Average RTT Measurements from Run 2 62

Table 4.4 Average RTT Measurements from Run 3 62

Table 4.5 Z-statistic and Percent of Change Results from Run 1 of Latency Experiment 63

Table 4.6 Z-statistic and Percent of Change Results from Run 2 of Latency Experiment 63

Table 4.7 Z-statistic and Percent of Change Results from Run 3 of Latency Experiment 63

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ACK	Acknowledgement
ARP	Address Resolution Protocol
AS	Autonomous System
BGP	Border Gateway Protocol
CDF	Cumulative Distribution Function
CDN	Content Delivery Network
CPU	Central Processing Unit
DDoS	Distributed Denial of Service
DOD	Department of Defense
DNS	Domain Name System
EUI	Extended Unique Identifier
FIFO	First In First Out
FIN	Finish
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
ICMPv6	Internet Control Message Protocol Version 6
IDS	Intrusion Detection System
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPv4	Internet Protocol Version 4

IPv6 Internet Protocol Version 6

ISN Initial Sequence Number

IW Initial Window

LAN Local Area Network

MAC Media Access Control

MD5 Message Digest 5

MSL Maximum Segment Life

MSS Maximum Segment Size

NIC Network Interface Card

NPS Naval Postgraduate School

OS Operating System

OUI Organizationally Unique Identifier

PDF Probability Distribution Function

RA Registration Authority

RST Reset

RTT Round Trip Time

SACK Selective Acknowledgement

SMTP Simple Mail Transfer Protocol

SSH Secure Shell

STL Standard Template Library

SYN Synchronize

TCP Transmission Control Protocol

UCSD University of California San Diego

UDP User Datagram Protocol

UML Unified Modeling Language

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

First of all, I want to thank my advisors, Rob and Justin, for all their guidance and valuable insight throughout this thesis process. It would have been impossible without them. I also thank Mark for all his insightful feedback. Secondly, I want to thank the SFS program and NPS for the opportunity to learn and grow in the cyber world, from one who did not even know what a "Hello World" program was, to one who is ready to join the cyber security workforce. Thirdly, I thank my mom and dad for their constant love, support, and encouragement, and for always reminding me that obstacles can be overcome with perseverance and continuous prayer. Last but not least, I thank God for giving me purpose and direction in life. To God be the glory.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Networks are constantly bombarded with malicious or suspicious network traffic by attackers attempting to execute their attack operations. One of the most prevalent types of traffic observed on the network is scanning traffic from reconnaissance efforts. For instance Yegneswaran *et al.* determined that worm activity constituted approximately 20%-60% of all intrusion attempts and observed as many as 3 million scans in a single day in their logs [1]. Dainotti *et al.* observed 20 million scanning probes from 3 million distinct source Internet Protocol (IP) addresses reaching the University of California San Diego (UCSD) Network Telescope due to the *sipscan* [2]. Rajab *et al.* collected three months' worth of Intrusion Detection System (IDS) logs and noted 1.5 billion connection attempts from approximately 32 million unique source IP addresses that originated from either compromised hosts or active scanners [3]. According to the EC-Council's Certified Ethical Hacker material, reconnaissance–one of the five phases in black hat operations–is the initial and longest phase, during which attackers perform non-intrusive passive network scanning among other actions to identify active hosts, operating systems, applications, and ports in a network [4]. After mapping out an organization's network structure, attackers follow up by using active scanning methods to discover weaknesses and potential vulnerabilities in the network infrastructure, such as open ports, services, and vulnerable applications. Because reconnaissance is a major phase in an attacker's operations, and because of the widespread and frequent nature of vulnerability scanning, efforts have been developed on the defenders' side to thwart or delay the attacker at this phase. One of the available network security defenses against worms and scanners is network deception.The high-level idea of network deception is to provide the illusion of vulnerable targets or to alter the network topology in such a way as to confuse attackers and divert their attention away from critical resources. Not only can deceptive strategies create believable targets, but some techniques can also actively influence an attacker's actions and decision making process [5].

Network tarpits are one example of network deceptive techniques that seek to slow automated scanning or confuse human adversaries [5]. These network tarpits are typically lightweight, low-cost applications that create sticky connections in the same way that physical

tarpits have been known to trap animals. They can be configured to use unused or inactive IP address spaces to create the illusion of a pool of hosts. The main goals of a tarpit are to penalize malicious connections by actively preventing data transfer over the connection as well as tying up the remote end's socket and exhausting the remote host's resources in maintaining this connection [5]. One of the obstacles tarpits face is the possibility of their discovery by advanced scanning techniques, which causes them to lose their deceptive advantage over an attacker. Attackers may be able to bypass or circumvent tarpits during their intelligence gathering phase. For example, *Degreaser*, an advanced tarpit scanning tool developed by Alt *et al.* has uncovered numerous network tarpits in the wild [5].

In their study, Alt *et al.* [5] develop an active probing methodology to detect tarpits based on two main signatures: Transmission Control Protocol (TCP) options fingerprints and TCP flow control behavior. This methodology is integrated into their open source tool, *Degreaser*, and used to perform an Internet-wide scan. Although organizations are reluctant to admit their use of tarpits for defensive reasons, Alt *et al.* discovered a total of 215,000 active fake IP addresses with subnetworks interleaved with real and fake hosts. Their work demonstrates the ease at which attackers and others can develop scanners to detect and defeat the use of tarpits based on the tarpits' current state and implementation [5].

1.1 Motivation

The main motivation for our work stems from two areas. The first is that network tarpits are powerful, important resources able to thwart network scanning during an attacker's intelligence gathering phase. Alt *et al.* note that even small blocks of tarpit addresses have greatly slowed automated scanning, and tarpits have even skewed network measure studies, e.g., the Internet census [5]. The second reason is that there has been a lack of current research available looking into improving the efficiency and effectiveness of network tarpits. Our work seeks to accomplish a couple objectives. We investigate heuristics to improve the tarpitting capabilities of the tarpit and better obscure them from state-of-the-art scanning tools and techniques. In addition, we develop and provide a modularized extensible tarpit platform called *Greasy* for researchers to build upon in the future.

1.2 Research Questions

We attempt to address the following research questions:

1. How can we mitigate network tarpit signatures to make them harder to detect, or more costly to detect, while retaining their defensive behavior?
2. How can we make network tarpits appear more like real hosts without incurring additional costs, like memory or bandwidth, to tarpit operations?
3. How can we leverage TCP retransmission mechanisms to improve the stickiness of the tarpit connection?

1.3 Summary of Contributions and Findings

Our work demonstrates that state-of-the-art tarpit scanning tools can be defeated. We make four primary contributions:

1. We identify distinguishing tarpit signatures and shortcomings of existing tarpit applications as uncovered by *Degreaser*, and implement improved features into a new tarpit application called *Greasy*.
2. We develop and provide a modularized extensible tarpit platform for future tarpit development.
3. We examine *Greasy*'s overall tarpit performance improvement compared to an existing tarpit application *LaBrea* [6] using a select set of metrics. We demonstrate *Greasy*'s ability to evade detection by *Degreaser*.
4. We run *Greasy* in production on an Internet-facing /24 subnet
5. We suggest future improvements to that may increase *Greasy*'s tarpit functionalities and capabilities in the future.

1.4 Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2 discusses background and related work in the area of cyber deception and network tarpits.

- Chapter 3 describes the requirements, improvements, design, and implementation of a new tarpitting application called *Greasy*.
- Chapter 4 provides and overview the experiments used to measure the levels of improvement in *Greasy* compared to existing tarpit applications. In particular, we measure *Greasy*'s ability to obscure itself from *Degreaser*.
- And finally, Chapter 5 concludes with highlights of experiment results and discusses future work.

CHAPTER 2:
Background

Among the onslaught of abusive and malicious attacks against networks, scanning attacks are crucial threats as they form the basis of other malicious operations. In order to infect a host in a network, an attacker frequently first scans the network (or the entire Internet) to find a reachable host that has some particular characteristic or exploitable vulnerability [7]. Intrusions by port scanning or self-propagating worms, e.g., CodeRed [8], Conficker [9], and Heartbleed [10] vulnerability scanning are examples of known scanning attacks [11]. Because network scanning is so pervasive, network administrators often apply network security defenses to impede scanning. Among these defenses are cyber deception techniques.

2.1 Cyber Deception

Cyber deception has contributed to the successes of both offensive and defensive operations. For defense in particular, the idea of cyber deception is to provide attackers the illusion of vulnerabilities so as to delay, halt, respond to, or gather information about suspicious activities or attacker behavior [12]. Most defensive deception strategies are only peripherally deceptive, meaning the objective is to create the facade of a real system, rather than deceiving the attacker through misinterpretation of data content [13]. Deceptive tactics force attackers to increase effort and costs to breach a system by causing attackers to spend time evaluating attack operations on fake resources, thus slowing their progress while gathering intelligence and increasing the likelihood of their discovery [5]. For instance, *moving target defense* systems increase the attacker's difficulty to exploit a vulnerable system by varying the target's attack surface or system behavior, thereby actively influencing the attacker's actions through deceit [14].

One deceptive strategy is to falsely plant information. The key to this deceptive strategy is to identify the attacker's targeted resource, plant falsities for the resource and underlying environment, determine desired consequences for the attacker's interaction with the fake system, and ensure the fidelity and sustainability of the resource as the attacker interacts with the system [13]. Another deceptive strategy is camouflage, which creates an artificial

disguise for a system or object to make it harder to find or identify. Noise injection is one camouflage technique that introduces additional noise to reduce the signal to noise ratio, making it more difficult to compromise a particular system based on signal indicators. Another technique is to reroute attacks in order to deflect them from the most critical systems [13]. Examples include lightning rods which distract attackers away from real targets to focus on interesting fake targets, jails that encapsulate attackers in order to conduct information gathering and analysis of attack methods, and shunts that redirect attacks around potential targets [13]. A popular tool used to reroute attacks is a network honeypot.

2.2 Honeypots

Honeypots are security devices, services, or resources in the network that detect and analyze intruder activities by creating false targets. The main idea behind a honeypot is to redirect potentially malicious traffic to a specially crafted machine. The machine is the "bait" with which intruders may interact [15]. Because honeypots have no production value, authorized activity, and often inhabit network address spaces where minimal or no legitimate traffic should exist, any interaction with the honeypot is likely an unsolicited probe, scan, or attack [16]. Honeypots can capture a wealth of information regarding suspected malicious behavior, including methods to gain unauthorized access, targeted ports, and services attackers attempt to exploit. Honeypots often operate in conjunction with other IDSs, and serve to extend IDS functionality by detecting and responding to attacks not identified by other detection systems [15].

2.2.1 Level of Interaction

Different information can be acquired by the honeypot depending on the honeypot's level of interaction.

High-Level Interaction

High interaction honeypots are used for more in-depth analyses, such as an investigation of tools, channels, or methods of attacks [17]. They replicate full system functionality and allow the attacker to fully interact with the system, and potentially compromise it to launch further attacks [15]. Though the data acquired from high interaction honeypots is quite

valuable as they allow researchers to study attack methods and attacker behavior, they are less prevalent compared to low interaction honeypots because the risk of an attacker negatively affecting other devices on the network is quite high. An attacker could simply use the honeypot machine to infiltrate other systems on the network, or lower system performances by decreasing bandwidth through generating large amounts of traffic [15]. Research honeypots typically have a high level of interaction with attackers because they are purposed to gather information about black hat techniques and tactics as well as monitor attacker behavior [15], [16].

Low-Level Interaction

Low-interaction honeypots emulate certain services, rather than a complete system, and as such are confined to respond only to specific services or partial implementations [18]. For instance, they may only emulate the transport layer protocol, and are unable to gather information on the application layer. Despite this limitation, low interaction honeypots are the most prevalent type of honeypot used today [19] and because low interaction honeypots offer limited services to the attacker and are at a low risk of being compromised, unlike their high interaction counterparts, they are easier to deploy and maintain [16], [20]. In addition, low-interaction honeypots are typically chosen to handle scanning and other attack traffic over high-interaction honeypots because they are much less costly (i.e., in terms of Central Processing Unit (CPU) time and bandwidth, for instance).[1] They are designed to lure an intruder with one or more exploitable vulnerabilities, and establish and capture the first few transactions of a conversation between the malicious endpoint and the honeypot. These honeypots allow for quantitative, less in-depth, information gathering used to identify patterns of port scans and worm propagation, passively detect spam activity, or delay malicious behavior [17]. Production honeypots traditionally fall under this category of interaction. They are purposed to delay, detect, or respond to attacks [16].

[1] Because of the benefits and shortcomings of both low-interaction and high-interaction honeypots, recent work has investigated the use of hybrid honeypot systems that utilize the strengths of both to dynamically adapt to organizational needs. Thus, low-interaction honeypots are deployed against simple attacks and act as a first layer of defense, while more sophisticated attacks are redirected towards high-interaction honeypots [21].

Medium-Level Interaction

Medium interaction honeypots are more advanced than low interaction honeypots, and less advanced than high interaction honeypots, however they combine advantages of both types [22]. Like low interaction honeypots, these honeypots do not implement complete application services or real operating system environments. A key component of these honeypots is their application-layer virtualization, which is what allows these medium-interaction honeypots to analyze malware similar to high-interaction honeypots but with an isolated environment and a small set of services. These honeypots lure attackers to execute their exploits by providing fictional vulnerability responses that an attacker expects to see [22]. After receiving the shell code for offline analysis, the honeypot emulates actions that the shell code would have performed on a real system. These honeypots do have limitations. Like high-interaction honeypots, these systems are also considered high risk because of the level of interaction with an attacker. They are also complex, time consuming to deploy, and require great knowledge and expertise to create [22].

2.3 Network Tarpits

Network tarpits are a type of low interaction, production honeypot that seek to frustrate and confuse an intruder, as well as slow automated traffic scanners [5]. They are modeled after the concept of physical tarpits that trap animals in sticky tar, and as such are sometimes called "sticky" honeypots. Tarpits attempt to trap potentially malicious incoming TCP connections through TCP flow control mechanisms – discouraging the remote end from disconnecting, actively preventing data transfers, and consuming the attacker's resources (e.g., CPU time, memory, and bandwidth); all these work to slow the scanner and penalize these malicious connections [5]. Existing network tarpits are deployed at both the transport and application layers of the TCP/IP stack. They may be configured to use unused or inactive IP addresses in a subnetwork to create the illusion of a large pool of available hosts and impersonate real hosts by responding to TCP, User Datagram Protocol (UDP), or Internet Control Message Protocol (ICMP) probes that interact with these fake hosts [5]. One of the most widely known network tarpits is *LaBrea*, developed by Tom Liston [23] in response to the *CodeRed* worm of 2001 [18].

2.3.1 *LaBrea*

LaBrea's main objective is to slow the propagation of Internet worms, by trapping the worms to delay their attempts of infecting other devices or systems [24]. *LaBrea* accepts connections to unused IP addresses on a local network, and responds to incoming traffic destined to these IP addresses, forcing the remote end to consume resources on a fake connection until it times out [24]. *LaBrea* has two main functions: dynamically identifying unused IPs using Address Resolution Protocol (ARP) and tarpitting connections using TCP flow control manipulation.

Identifying Unused IP Addresses

To determine which IP addresses to use for tarpitting, *LaBrea* either hard captures a specific list of IP addresses, or listens promiscuously for unanswered ARP requests, which signify unused or unoccupied IP addresses [24]. *LaBrea* takes advantage of the Layer 2 ARP requests and responses in order to locate these unused Internet Protocol Version 4 (IPv4) addresses and impersonate hosts. ARP provides a mapping between network and physical addresses. In order for two machines to communicate across an Ethernet Local Area Network (LAN) each machine must have two unique identifiers: a layer 3 32-bit IPv4 or 128-bit Internet Protocol Version 6 (IPv6) address, and a layer 2 48-bit Media Access Control (MAC) address. The source device endpoint must know the destination endpoint's MAC address in addition to the destination IP address in order to send a packet to the destination device endpoint. For IPv4 addresses in particular, if the destination IP/MAC address pair is not stored in the source endpoint's cached ARP table, it must send a broadcast ARP request to devices in the LAN to find the MAC address of the destination endpoint [25]. If an ARP reply successfully returns with a corresponding MAC address, that IP/MAC pair will be stored in the router's cache.

Else if the destination device does not exist, the packet will be dropped, the router will send back an ICMP "host unreachable" message, and no entry will be stored in the ARP table for that pair [26], [27]. Mechanisms have been used to flush stale ARP cache entries using ARP cache timeouts. One of these methods involves periodic active probing of the remote host using ARP requests. If these requests remain unanswered after N successive probes (with retransmissions as needed), the entry is deleted. N is typically 2 probes, but is configurable [28], [26]. Additionally, ARP is used to resolve IPv4 address conflicts.

Before using an IPv4 address, a host must check to see if that IP address is in use by broadcasting an ARP request to that particular IP address. The host sends PROBE_NUM packets spaced PROBE_MIN to PROBE_MAX seconds apart. If ARP requests remain unanswered after ANNOUNCE_WAIT seconds, then the IPv4 address may be used safely by the host [29]. PROBE_NUM, PROBE_MIN, PROBE_MAX, and ANNOUNCE_WAIT values are configurable.

LaBrea monitors for consecutive unanswered ARP requests without intervening ARP replies [6]. If after the timeout (a value that can be configured in *LaBrea* but defaults to 3 seconds) *LaBrea* still sees unanswered ARPs for a particular IP address, *LaBrea* will capture the IP and send an ARP reply to the requesting host with a fake MAC address set to (00:00:0f:ff:ff:ff). There is no significance for this MAC address other than that it is bogus and does not actually exist; in addition, it is statically applied to all packets sent on the LAN by *LaBrea* [24], [6]. This method even works for switched LANs. Typically the broadcast ARP requests are visible to all ports on a hub device because data is flooded to all ports on the device. Switch devices, on the other hand, maintain an address table of associated hardware addresses and ports [30]. Thus, one port may never see traffic that another port may see, and *LaBrea* may never see ARP replies for ARP requests on the LAN. *LaBrea* overcomes this by sending a "mirrored" ARP request of any ARP requests it sees, which allows *LaBrea* to monitor for potential ARP replies in a switched environment [6]. The method discussed describes *LaBrea*'s soft-capture mode, where *LaBrea* captures IP addresses dynamically. In addition, *LaBrea* also offers a hard-capture method and auto-hard-capture method to capture IP addresses to use as tarpit hosts. The hard-capture method states that after *LaBrea* captures an IP address, it does not need to wait for the timeout before sending an ARP reply. The auto-hard-capture mode states that all non-excluded[2] and non-hardexcluded[3] IP addresses should be automatically hard-captured.

TCP Flow Control Manipulation and Tarpitting Functionality

TCP flow control [31] manipulation is the heart of *LaBrea*'s TCP level tarpit deception, and the key to trapping an adversary in an open TCP connection. Among the objectives of TCP

[2]Excluded IP addresses are those in the LAN that *LaBrea* should never capture [6]

[3]Hardexcluded IP addresses only apply if the hard-capture mode is set; this instructs *LaBrea* to always wait for the ARP timeout and never hard-capture these IP addresses [6].

in network communication is to accurately match the transmission rates of data transfer between two end devices [31]. Its main concern is to prevent the remote end from receiving data faster than it can process. For instance, a larger window size could cause buffer overruns, packet loss in routers with smaller buffers, or congestion, if the receiving end is unable to handle packets sufficiently. This is wasteful as the remote end must retransmit those lost bytes of packets [32]. TCP uses a 16-bit window field in its header to implement flow control. The receiver uses this field to indicate how many bytes of data the receiver is willing to accept and process from the sender, thereby establishing the max transmission rate [31]. This information is sent to the sender through Acknowledgement (ACK) packets. Included in this transaction is an implicit trust arrangement between the sender and receiver, which is exploited by *LaBrea* [25]. In order to lock a client in a fully established TCP connection, *LaBrea* uses the window field in the TCP header to manipulate flow control. Normally, TCP can advertise an Initial Window (IW) size smaller than 10 segments. The upper bound for the IW is defined as min (10*MSS, max (2*MSS, 14600)), where MSS is the maximum segment size of the TCP segment. In addition, the Synchronize (SYN)+ACK and ACK packets of the TCP 3-way handshake should not increase the IW size [32]. In persistent mode, a tarpit advertises a zero window size in ACK packets sent to the sender. This indicates to the sender that the tarpit's receive window may be closed indefinitely, but as long as the tarpit continues to send ACKs in response to the sender's window probes, the sender must allow the connection to remain open [33].

In persistent mode, *LaBrea* never increases the window size, keeping the client in a persistent connection. As soon as an ACK packet with a zero window size is received, the sender starts the TCP persist timer, and sends a window probe everytime the timer expires (i.e., a timeout range of 5 to 60 seconds) to query the receiver to see if the window size has increased. It is important to note that TCP does not give up sending window probes, which can lead to resource exhaustion [34]. This can create a window probe deadlock in which the window probe exchange continues indefinitely, unless the remote end has implemented an alarm to break out of the deadlock. The attacker that initiated the TCP connection may send a Finish (FIN) packet to terminate the TCP connection after a period of time has elapsed or limited number of window probe retransmissions are exhausted with no increase in window size from the receiver [35]. The specific time-frame in which a remote end terminates the connection is implementation-specific. One of the ways TCP closes a connection is by hav-

ing one of the endpoint users initiate the connection termination by sending a FIN packet to indicate that user is finished sending data. The user then enters the FIN-WAIT[4] state, and waits for the remote end to ACK the user's FIN and send a FIN packet of their own. If the user does not receive a FIN+ACK packet from the other end before the FIN-WAIT timeout expires (i.e., 2 Maximum Segment Life (MSL)), then the connection will timeout and close [31]. FIN packet *LaBrea* ignores this request to end the connection, and maintains the connection until the FIN-WAIT period expires, thus exhausting the client's socket resources to maintain this connection state. *LaBrea* also offers a non-persistent mode in which all packets received by the client after the TCP handshake is established are just ignored, and the TCP connection terminates after a MSL of 2 minutes [5], [31]. Although *LaBrea*'s persistent connection is able hold a connection indefinitely versus several minutes for a non-persistent connection, the persistent connection does contain more traffic overhead than the non-persistent connection as *LaBrea* has to send packets back to the attacker. The overhead is on the order of around 1215 bytes/hour according to Haig [27].

TCP Retransmission Manipulation

Aside from persistent and non-persistent modes of tarpitting, we can also leverage the use of TCP retransmissions to disrupt the flow of a TCP connection. TCP has several mechanisms to indicate packet loss and out-of-order packets to the opposite endpoint. Two of the mechanisms are duplicate ACKs and partial ACKs. Duplicate ACKs are normally sent by the receiver to notify the sender that an out-of-order data segment arrived. The sender usually views duplicate ACKs as an indication of network problems. For instance, duplicate ACKs are usually sent as a result of dropped segments, data segments re-ordered by the network, or replication of ACKs by the network. The sender remedies the situation by using a fast retransmit algorithm to detect packet loss. The sender's algorithm waits for the arrival of three duplicate ACKs without any other interleaving packets, as the indicator of packet loss, and immediately retransmits the missing segment [36]. Similarly, partial ACKs signal packet loss to the sender, and cause the sender to retransmit "lost" packets [37]. This mechanism can be used by a defender to emulate packet loss and misordered packets to force retransmissions by the sender and disrupt the connection.

[4]The FIN-WAIT period is the time an endpoint must wait for a TCP connection termination request from the remote end, or an acknowledgement by the remote end of a termination request issued by the endpoint [31].

2.3.2 Limitations of Tarpits

The usefulness of tarpits, like all honeypots, lies in their deceptive abilities, i.e., their ability to deceive an attacker into believing some fiction is true. However, sooner or later, if attackers look carefully for signs of deception, he or she will be able to detect signatures [38]. The stealthiness and undetectability of a tarpit are important characteristics key to the success of its functionality. Attackers are developing ways to detect tarpits and other honeypots in an effort to circumvent them. The existence of these detection tools, such as *Degreaser* [5] is the main motivation to improve the current state of tarpits, and will be further discussed in Section 2.4.1.

2.4 Related Work

2.4.1 Honeypot Detection

Attackers are developing new tools and techniques to detect the presence of tarpits and other honeypots, thereby avoiding these traps and making these tarpits useless.

Degreaser

Alt *et al.* developed *Degreaser* [5], a fingerprinting tool designed to scan a list of subnets and remotely detect tarpits based on tarpit-like signatures [5]. Alt *et al.*'s motivation for developing *Degreaser* stems from defensive security objectives. They empirically investigate how tarpits pollute network measurement studies, as well as note the negative impact that even small blocks of tarpit address spaces have on automated scanners. Thus they leverage the limitations of tarpits due to flaws in their deception, to develop heuristics for tarpit detection. Alt *et al.* focused on iptables TARPIT [39] plug-in and *LaBrea* in the development of *Degreaser*. They discovered two defining tarpit characteristics. The first is a small TCP window size where *LaBrea* typically advertises a default TCP window size of 10 bytes and iptables TARPIT plug-in returns a window size of 5 bytes. Both *LaBrea* and iptables TARPIT plug-in manipulate TCP flow control mechanisms to tarpit a remote host. Both tarpit applications either hold non-persistent or persistent connections. *Degreaser* looks for window sizes less than 20 bytes as the first indicator. The second characteristic is the lack of TCP options in TCP packets sent by both tarpits, which is fingerprinted by *Degreaser*. In addition to deploying degreaser against known, ground-truth tarpits, Alt *et al.* also perform

an Internet-wide scan to deduce the existence of tarpits in the wild. They discover 107 different subnets composed almost entirely of tarpit hosts or consisting of a combination of tarpit and real hosts. Their findings indicate that existing tarpit applications may be easy to detect and defeat [5].

Service Exercising

Mukkamala *et al.* discuss a technique called service exercising [40], that leverages the incomplete feature set of low interaction honeypots and TCP/IP fingerprinting techniques to identify honeypots in the wild. One technique described is timing analysis which uses a stream of ICMP echo requests to measure latency of packets. Their results show that timing analysis can clearly distinguish honeypots from real systems. In addition they isolate and test certain uncommon features or operations that low interaction honeypots may not have implemented in order to determine whether the system is a honeypot or a legitimate system [40].

2.4.2 Other Existing Tarpits

Honeyd

Honeyd is a virtual honeypot that offers a tarpit option among a slew of other services. It assumes the identity of any unused IP address to create a honeynet of hosts, with the goal of forwarding all traffic interacting with these unused IPs to the *Honeyd* honeypot [24]. This honeypot has the ability to emulate different services and operating systems, thereby providing a platform for an active response model used to analyze both non-automated and automated exploits [18]. *LaBrea* differs from *Honeyd* in that *LaBrea* is focused on thwarting an attacker's reconnaissance, whereas *Honeyd* is used more like a traditional research honeypot and detection tool used to detect attacks and unauthorized activity [41]. Ruvalcaba claims that *LaBrea* and *Honeyd* are two different but complementary security tools that could be combined to create a tarpit-honeypot system, with *LaBrea* as the first line of defense and *Honeyd* as the tool for more in-depth analysis of the malware [18]. In addition, despite offering numerous services on top of the tarpitting functionality, *Honeyd's* inherent statefulness limits its scalability [18]. *Greasy*'s design attempts to avoid this overhead.

SMTP Tarpit

The Simple Mail Transfer Protocol (SMTP) Bulk Mailer tarpit targets the TCP port for SMTP and slows traffic by reducing the frame size of a packet to the bare minimum, simulating packet loss, and reducing the window size. The goal of this tarpit is to give the attacker the impression of a very slow server to discourage spam messages from coming through the network. Though this tarpit drastically reduces the transmission speed, it also has the disadvantage of increasing network traffic due to its protocol overheads [42].

In addition to TCP level SMTP tarpitting, other implementations take advantage of SMTP continuation lines to delay bulk mailers on the application level [43]. RFC 821 [44] states that SMTP reply messages longer than a single line must be broken up into a multi-line, (i.e., continuation lines), reply. The format is "<Reply Code>: -". These application layer SMTP tarpits send lots of fake continuation lines, slowing down the bulk mailer connection. But they are not very effective. But these tarpits are not very effective because bulk mailers connect to multiple mail servers at a time, mail servers accept multiple emails in a single connection, and these tarpits are only able to delay one connection to a mail server at a time [43].

HTTP Tarpit

The Hypertext Transfer Protocol (HTTP) tarpits are used to prevent harvesters from collecting email addresses that are ultimately used in a spam mail attack. The goal is to create random webpages that contain random links, thereby increasing the total amount of webpages these harvesters have to comb through in order to collect legitimate email addresses. This will delay the malicious email acts of the adversary. Unlike the SMTP Bulk Mailer tarpit that protects a single mail server from spam, HTTP tarpits prevent spam more on the global scale since they actively prevent the harvester from moving onto other webpages to collect email addresses [43].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Requirements and Design

This thesis focuses on accomplishing two main objectives. The first objective is to better obscure network tarpits from scanners such as *Degreaser*, while still retaining their tarpitting behavior. The second objective is to develop a tarpit application with a modular multi-threaded architecture that can be easily extended to handle other types of packets and protocols in the future, e.g., IPv6 and application-layer tarpitting capabilities further discussed in Chapter 5. This chapter discusses the design requirements, development, and implementation of a new network tarpit called *Greasy*.[5]

3.1 Requirements

Based on the literature review and lessons learned from *Degreaser* we consider the following important areas to incorporate in the design of *Greasy*:

- Deterministic Randomness
- Maintaining Minimal State
- Balance of Costs

3.1.1 Deterministic Randomness

One area of importance is deterministic randomness, which introduces realistic varying behavior. Randomness plays a key role in protecting the deceptive quality of a tarpit because it prevents the tarpit from leaving behind footprints or signatures. However, from the perspective of a given adversary, the behavior should not appear random, but should present the consistent appearance of a realistic remote host – otherwise the random behavior itself becomes a signature of the deception in place. For instance, if two remote hosts communicate with the same tarpit, we want the tarpit host to return the same packet information

[5] *Greasy* is a new tarpit application developed in response to Alt *et al.*'s [5] findings, that combines several algorithms from *LaBrea*, new heuristics for random tarpitting behavior, and tarpit signature mitigation techniques in an effort to better obscure its detectability from *Degreaser* and other potential tarpit scanners.

(e.g., MAC address, TCP options, and available port numbers, among others) to the two remote hosts.

3.1.2 Maintaining Minimal State

Another key area is maintaining minimal state using low overhead hashing techniques to mimic state behavior without actually incurring state. Currently the simplicity of existing tarpits is their statelessness, but implementing new tarpit heuristics may incur the cost of state, which may decrease tarpit performance due to large overheads (e.g., packet queuing congestion, increased time utilization, or exhausting memory). For this reason, we deploy SYN cookie-like [45] heuristics to minimize state maintenance in *Greasy*.

Minimal State Using SYN Cookies

SYN cookies [45] are used to combat resource exhaustion in the event of a SYN flood-based attack against a system. The cookie is a known public hash with a secret key, used to verify a TCP connection, and is composed of the timestamp, maximum segment size stored by the server, server IP address and port number, and client IP address and port number. Upon receipt of a SYN packet, the system generates the hash value and discards the SYN queue entry. This hash value is then sent as the initial sequence number in a TCP SYN+ACK packet during the TCP handshake. If the server receives a TCP ACK to this packet with an ACK value of the server's hash plus one, then the server can verify that the connection is legitimate. By deploying the SYN cookie, the system does not need to maintain any state until the connection is verified, which minimizes the system's burden of exhausting resources to remember half-open TCP handshakes [45]. We deploy a similar method throughout *Greasy*'s design that is further discussed in Section 3.3.

3.1.3 Balance of Costs

A third area of concern is balance of costs in our modifications. It is important to note that there exists a basic tension in our efforts to improve the network tarpitting capabilities in *Greasy* and decreasing its detectability. For instance, in order to mitigate the tarpit's distinguishing window size signature, we will not solely change the window field to a larger size in order to evade detection. Doing so may allow normal data transfer to occur,

allowing more traffic to come onto the tarpit's network, and negating the main functions of a tarpit. Another example is the notion of state, as discussed in Section 3.1.2. Performing these improvements on a new tarpit will impose a cost on the attacker as the attacker must increase his efforts of detecting tarpits and may suffer a greater loss of time and resources as a result of the increased stickiness of the tarpit. This may, however, also incur a cost on the defender as well due to the increased overhead (i.e., memory or time) required to deploy a more advanced tarpit.

3.2 Improvements

We focus our tarpit improvements on several key areas based on recommendations from *Degreaser*'s findings. Alt *et al.* [5] identify discriminating traffic characteristics of network tarpits deployed in their tarpit detection heuristics, as well as signatures that were interesting but proved unfeasible or unreliable for inclusion in *Degreaser*'s design; we incorporate improvements that combat signatures both utilized and not utilized in *Degreaser*'s heuristics. We divide the types of improvements into two main categories: improvements which are trivial to implement and should not affect the tarpit's overall performance, and improvements which may cause an imbalance between tarpit performance and detection.

3.2.1 Light-Weight Enhancements

The following are improvements that will increase the difficulty for an attacker to fingerprint the tarpit without affecting its overall performance.

- Inclusion of TCP options
- MAC address generation
- Number of Open Ports
- Limiting the Number Responding IP addresses

Inclusion of TCP Options

The TCP options [46] are appended to the end of the TCP header and are used to introduce add-ons or additional features to enhance the TCP protocol. Some options include: Maximum Segment Size (MSS), Window Scale, Selective Acknowledgement (SACK), and

Timestamps. *Degreaser* identifies tarpit hosts by observing the lack of TCP options appended to the TCP header of packets sent by those hosts [5]. We mitigate this signature by appending a randomly determined set of TCP options to the end of the TCP header without added costs to tarpit performance. *Greasy* contains an array of five sets of TCP option combinations; each tarpit host is assigned a particular set of TCP options to append to the TCP header of packets they send using a custom hashing function as shown in (3.1).

$$index = ((targetIPaddr * 59) \oplus targetPort)\%5 \tag{3.1}$$

The function is a modified version of [47] used to hash a 5-tuple IP, port, and protocol numbers to identify flows. Our function uses the target IP address and port number from the incoming TCP packet, and an arbitrary prime number (i.e., 59 in our case [48])[6] to create a random hash value. The hash value modulo 5 determines the index into the array of TCP option sets. By using this function, we always pair TCP packets from a particular tarpit host/port with the same set of TCP options without maintaining state and avoid the costs mentioned in Section 3.1.2. The TCP options combinations were determined based on documentation from *p0f* [49], a passive OS fingerprinting tool, as well as through examination of TCP packets on my LAN. We did not include SACK [50] as an option, as it would impede our goal of causing the remote end to retransmit the entire data segment vice small parts of the missing segments.

MAC Address Generation

The MAC address is a physical address assigned to network interfaces for most Institute of Electrical and Electronics Engineers (IEEE) 802 devices [51]. We focus on the universally[7] administered 48-bit EUI-48 address[8] [52]. Alt *et al.* [5] observed that *LaBrea* uses a hard-coded Layer-2 Ethernet MAC address (00:00:0f:ff:ff:ff) for all responses regardless of its

[6]The prime number is used to produce a prime displacement hash function [48] that promotes uniqueness

[7]MAC addresses are either administered locally or universally. The first 3 octets in an universally administered MAC address indicate the identifier of the device manufacturer, or Organizationally Unique Identifier (OUI) assigned by the IEEE Registration Authority (RA). Those 3 octets can be overwritten by a network administrator if the address is administered locally.

[8]Of the three EUI-48 address schemes, we use the concatenation of a 24-bit OUI and a 24-bit extension identifier assigned by the organization that belongs to the OUI, which is the addressing scheme for OUIs of large companies [52].

physical network adapter address. However, Alt *et al.* did not include this signature as part of *Degreaser*'s heuristics because they focused on finding remotely identifiable tarpit characteristics, whereas the MAC address is only discernible inside the LAN containing the tarpit host [5]. We decide to improve upon this feature for completeness, since our concern lies with adversaries attached to the same LAN as *Greasy*, as well as those scanning remotely. The MAC address sent in the ARP reply and in other packets sent by Greasy is generated using a deterministically random method. The upper 3 OUI octets is one OUI out of an array of thirteen company OUIs. The thirteen OUIs represent identifiers from each of the top thirteen device manufacturers (e.g., Apple, Asus, Cisco, Microsoft, and HP, among others), which we determined based on information provided by the IEEE OUI website [53], as well as the number of OUIs allocated to each company. The index into this array is determined by taking the target IP address modulo 13, as shown in (3.2).

$$index = targetIPaddr\%13 \tag{3.2}$$

The array of OUIs is listed in Table 3.1:

Table 3.1: The List of OUIs and Corresponding Index into *Greasy*'s Array of OUI Used to Generate the Upper 3 Octets of a Tarpit Host's MAC Address

Array Index	OUI	Company
0	0c:15:39	Apple
1	d8:50:e6	Asus
2	7c:69:f6	Cisco
3	a4:77:33	Google
4	8c:dc:d4	HP
5	98:be:94	IBM
6	6c:a1:00	Intel
7	ec:3e:f7	Juniper Networks
8	2c:54:cf	LG
9	30:59:b7	Microsoft
10	00:14:4f	Oracle
11	1c:66:aa	Samsung
12	fc:0f:e6	Sony

The lower 3 octets represent the random 24-bit identifier assigned by the organization of the OUI. To create this 24-bit identifier, we first take the Message Digest 5 (MD5) [54] hash of

the target IP address that produces a 32 character string, and then use the upper 3 octets of this hash for the lower 3 octets in the MAC address of a particular tarpit host. In this way, all remote hosts that converse with the same tarpit host should see the same MAC address associated to the tarpit host. Although the OUI portion could potentially be a signature, it is only detectable to an adversary on the LAN. This feature is an improvement from *LaBrea*'s statically assigned MAC address 00:00:0f:ff:ff:ff, as it provides a more realistic MAC address.

Number of Open Ports

TCP has 65,535 possible open ports. Ports 0-1023 are well-known ports, 1024-49151 are registered ports, and 49152-65535 are private or ephemeral ports. Certain ports, such as those within the well-known ports range are open for long periods of time, but others are usually only expected to be open for a short amount of time. The EC-Council considers persistent ephemeral ports as suspicious open ports because ephemeral ports are usually only short-term ports. Any ephemeral port left open for extended periods of time is suspicious, and similarly, it is rare that real live hosts have all 65,535 ports open at the same time [55]. In their Internet-wide scan, Alt *et al.* [5] observe that many hosts answer all TCP ports, and infer that hosts that respond to all TCP ports is indicative of a tarpit behavior[9]. They left this signature as a last resort to disambiguate samples of possible tarpit hosts and did not include this as part of *Degreaser*'s scanning heuristic because of the exponential (i.e., 2^{16}) number of probes that would be required to scan for open ports for a single host. This would be especially costly if executed at the Internet scale [5]. We decided to mitigate this signature, nonetheless, because we want to ensure that all potential signatures are removed from the tarpit. *Greasy* contains an array of five different sets of open ports; each tarpit host is assigned a particular set of open ports using the same hashing function shown in (3.1) to determine the index number into the open ports array. Each set consists of seven to ten well-known ports [56]. Using this method, we are able mitigate this signature without maintaining any state.

[9]By default *LaBrea* has a list of always open ports, and also dynamically opens ports based on the amount of activity from incoming connections to a closed port [6]. The only way to turn off ports is by manually excluding ports in *LaBrea*'s configuration file. Results from *Degreaser*'s scan may indicate that many hosts running *LaBrea* do not have any excluded ports.

Limiting the Number of Responding IP Addresses

Alt *et al.* [5] initially considered high-occupancy subnets as an indicator of tarpit hosts [5]. Normally only a small fraction of address space is actually utilized by real hosts at any given time, particularly the larger subnets like /16s [57]. However, they determined that this was not a good selection criterion because many of the /24 subnets they investigated [58] actually belong to Content Delivery Networks (CDNs) and other web-hosting services that do have fully occupied subnets. We decide to implement improvements for this signature based on results from their Internet-wide scan using *Degreaser*. From their Internet-wide scan, Alt *et al.* [5] note their discovery of subnets completely full of fake tarpit IP addresses, as well as subnets composed of a mix of tarpitting and real IP hosts. For instance, around 50% of all /24 subnets are filled with 95% or greater of IP addresses that exhibit tarpit-like behavior, 60% of /22 and /23 subnets are composed of 95% or more tarpit hosts, and two of the /16 subnets are composed of more than 99% tarpit hosts [5]. *Greasy* subverts the high subnet occupancy trait by incorporating a method to configure the number of responsive IP addresses within a subnet to mitigate its detection in the wild. We provide the user with the flexibility of determining the percent of responsive IP addresses in the configuration file (Section 3.3.4). To determine if an IP address falls within the fraction of IP addresses that should respond, we first take the MD5 hash of the target IP address. We then take the lower 8 hexidecimal digits of the hash and divide this by the baseline hex value "0xffffffff" to determine a percentage value. If this value is less than or equal to the percentage value noted in the configuration file, then *Greasy* will capture this IP address. We chose to only keep the lower 8 hexidecimal digits of the hash rather than its entirety because we wanted to minimize the search space. This improvement does not add additional operating costs.

3.2.2 Improvements with Potential Impact to Performance

The following are improvements that may affect tarpit performance and detection.

- Random Legitimate Window Size
- Heuristic for Random Tarpitting Behavior

Random Legitimate Window Size

One of the distinguishing tarpit characteristics detected by *Degreaser* is a small TCP window size, i.e., 20 bytes or less. We attempt to mitigate this signature by implementing a configurable maximum TCP window size determined by the user. The configurable TCP window size is an attempt to thwart the small TCP window size signature by establishing a random legitimate window size. The maximum window size value is established by the user in the configuration file (Section 3.3.4), and the window size for a particular TCP packet is calculated by using rand() % THROTTLE_SIZE, with the current time as the seed for rand() to create the pseudo-random number. We use a random window size as a opposed to choosing from a set of candidate window sizes because we want to mitigate any possible signatures that could be attributed to the window size. By implementing this improvement we do run the risk of *Greasy* receiving more network traffic than intended and experiencing network congestion. However, this is an important signature to mitigate in order to deceive *Degreaser*, and our hope is that the increased incoming traffic will cause only minimal congestion.

Heuristic for Random Tarpitting Behavior

In an effort to improve the effectiveness of the tarpit, we introduce the idea of random heuristics for tarpitting. Unlike *LaBrea*, which either executes in non-persistent or persistent modes, *Greasy* randomly executes non-persistent and persistent modes, as well as adds two more tarpitting options–duplicate ACK and partial ACK modes. The latter two are important tarpitting techniques that encourage the remote end to send retransmissions and keep the connection open, thereby creating the illusion of a legitimate but congested connection. *LaBrea* only allows one of the modes to execute at a given time. Persist mode is essential to tarpitting connections and delaying remote ends from attacking other more critical systems on the local network. However, the persistent zero TCPwindow size used in persist mode is too distinguishing of a trait and easily detected by scanning tools like *Degreaser*. For this reason we add additional tarpitting capabilities and randomize their usage to prevent adversaries from identifying a tarpitting fingerprint. The partial ACK and duplicate ACK modes, like the persistent mode [27], may carry traffic overhead, since *Greasy* will have to respond to the incoming packets to ensure the continuation of the connections. This will generate more traffic than just simply dropping packets in the non-

persistent mode. In addition, the duplicate ACK mode does maintain state, since it must remember the number of duplicate packets it has sent, as well as retain the same ACK value as those in the series of duplicate ACK packets sent. Despite these costs, we hope the improvements in the effectiveness of *Greasy*'s tarpitting capabilities and ability to deceive *Degreaser* will far out-weigh costs in tarpit performance.

3.3 Design

Greasy performs three primary functions: dynamically determining unused IP addresses on the network, impersonating real hosts by responding to ICMP and TCP packets, and tarpitting TCP connections by holding the connections open and actively preventing data transfer. These three functionalities are partitioned among three packet handler class modules, which are critical components of Greasy's overall modular design. We opted to build a modular tarpit application from the ground-up instead of building upon *LaBrea*'s open-source code because we wanted a platform that could easily extend the tarpitting functionality of the application to other types of packets or protocols in the future. Currently *LaBrea*'s code combines the handling of both TCP and ICMP packets into one large function with over 200 lines of code, making the code difficult to read and modify. In our design, we encapsulate the data representation and processing of different packet types as separate modules, which makes future extensions and modifications to *Greasy*'s code easier. We utilize *LaBrea*'s algorithms for dynamically capturing IP addresses, tarpitting TCP connections using non-persistent and persistent modes, and handling ICMP packets. *Greasy*'s modular design is discussed in detail in Section 3.3.1. Besides modularity, other notable features of *Greasy*'s design include its multi-threaded application to further modularize *Greasy*'s design by partitioning the different concerns of processing into separate tasks, and heuristics for deterministic random tarpit behavior, both of which will be further discussed in Section 3.3.2 and Section 3.3.3.

3.3.1 Modular Design Overview

Greasy's object oriented design partitions all functionalities among four new classes, a *PriorityQueue* template that orders packets in a queue by earliest arrival time in a First In First Out (FIFO) method, and *Greasy*'s *main* function, as shown in Figure 3.1. *Greasy*'s

main function contains an array of helper objects from the master class *GreaseMonkey*, whose job is to process the packets captured by the sniffer in *main*. The number of *GreaseMonkey* objects in the array is determined by the Standard Template Library (STL) function *std::thread::hardware_currency()*, which returns the approximate number of hardware thread contexts supported by the particular system on which *Greasy* is run. Each *GreaseMonkey* object in turn invokes one of the three handler classes to process the packet: *ARP_Handler*, *ICMP_Handler*, or *TCP_Handler*. In order to support the multi-threaded design, which will be discussed in Section 3.3.3 we implement a new mutex protected *PriorityQueue* template class. *Greasy*'s *main* function also calls the *config_parser.cpp* module to parse the configuration file (Section 3.3.4) and initialize variables and settings. But first, we discuss the features and algorithms of each packet handler class.

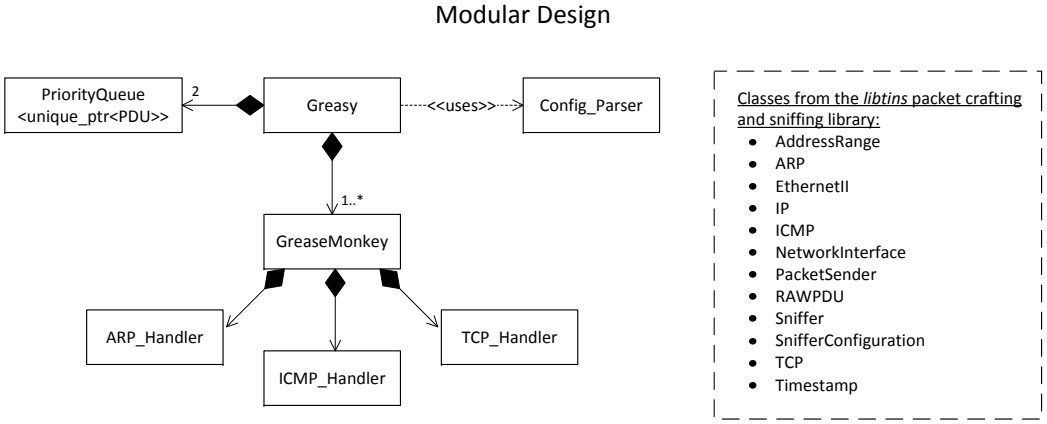

Figure 3.1: UML Class Diagram of *Greasy*. The black diamond signifies composition, or not-shared association, in which the class closest to the diamond has exclusive ownership over the classes at the other end of the arrow in a part-whole relationship. In the diagram *GreaseMonkey* contains the three packet handler classes. The numbers next to the *PriorityQueue* and *GreaseMonkey* objects indicate the number of instances of each object. The "1...*" next to the number indicates 1 or more instances of the object. The arrow from *Greasy* to the *config_parser* module represents a usage relationship, where *Greasy* uses functions from *config_parser* to parse the configuration file. The box on the far right are classes used by *Greasy* from the *Libtins* packet crafting library.

Algorithm 1 Algorithm for Dynamically Capturing IP Addresses

```
1: if Gratuitous ARP then
2:     Remove IP address from CAPTURED_IP list and add to REAL_IP list
3: end if
4: if ARP request then
5:     if Switched environment then
6:         Send mirrored ARP request
7:     end if
8:     if IP already captured then
9:         Send ARP reply immediately
10:    else if Soft_capture && (IP not in IP_IGNORE || REAL_IP lists) && IP is part of
       fraction of responsive subnet then
11:        if IP is entry in ARP_PENDING list then
12:            if Time val for entry is 0 then
13:                Reset val to current timestamp
14:            else if Max timeout reached then
15:                Reset val to 0
16:            else
17:                Capture IP and send ARP reply
18:            end if
19:        end if
20:    end if
21: end if
22: if ARP reply then
23:    Remove IP address from ARP_PENDING list and add to REAL_IP list
24: end if
```

3.3.2 Packet Handler Features and Algorithms

ARP_Handler

The ARP_Handler class module uses the following function to handle incoming ARP packets: *void handle_ARP_req(ARP *arp_recv, EthernetII *eth_recv)*. This function takes as parameters pointers to the ARP and Ethernet layers of an incoming packet, extracts required packet information (e.g., source and destination IP addresses, and source MAC address) required to build ARP request or reply packets, and sends outgoing packets back to its *GreaseMonkey* instance to be placed on the *send_q* (Section 3.3.3).

The *ARP_Handler* uses *LaBrea*'s algorithm for dynamically capturing unused IP addresses

Algorithm 2 Algorithm for function: *handle_TCP(...)*

```
1: if IP is in captured list && has corresponding open port then
2:     Call build_TCP_pkt(...)
3:     if build_TCP_pkt(...) returns true then
4:         Send TCP packet to GreaseMonkey instance to be added to send_q
5:     end if
6: end if
```

Algorithm 3 Algorithm for function: *build_TCP(...)*

```
1: Generate ISN and random window size
2: if TCP::FLAG == RST then
3:     Do not respond
4: else if TCP::Flag == FIN then
5:     Do not respond
6: else if TCP::Flag == FIN+ACK then
7:     Do no respond
8: else if TCP::Flag == SYN then
9:     Build TCP SYN+ACK packet and append set of TCP Options and return true
10: else if TCP::Flag == SYN+ACK then
11:     Send TCP RST
12: else if TCP::Flag == ACK then
13:     Verify incoming ACK is ISN + 1 and initialize entry for TCP session with tuple
14:     if Duplicate_bit set to 1 then
15:         Choose duplicate ACK tarpitting behavior
16:     else if Duplicate_bit set to 0 then
17:         Randomly determine tarpitting behavior out of the 4 modes and invoke
    tarpit_opt(...)
18:     end if
19: end if
```

in the LAN to be used as tarpit hosts. Like *LaBrea*, *Greasy* also offers the option of hard capturing IP addresses, for which the user can indicate in the configuration file (Section 3.3.4) particular IP addresses or ranges of addresses to capture. The other option is to soft capture IP addresses, or capture them dynamically. Using *LaBrea*'s method for listening for consecutive unanswered ARP requests spaced several seconds apart (default is 3 seconds) before capturing the corresponding IP address, *Greasy* listens for unanswered ARP requests for a given IP address within a configurable time frame. If the ARP requests are unanswered after a minimum time value, *Greasy* will assume this IP is unused, capture

Algorithm 4 Algorithm for function: *tarpit_opt(...)*

```
1: Enter switch statement
2: if Partial ACK then
3:     Store fraction of incoming payload as ACK value and return true
4: else if Persist then
5:     Set window size to 0 and return true
6: else if Non-persist then
7:     Drop packet and return false
8: else if Duplicate ACK then
9:     if Duplicate_bit set to 0 then
10:        Set bit to 1 and return false
11:    else if Duplicate_ACK bit is set then
12:        if Incoming packet sequence number > stored sequence number in entry then
13:            Set ACK as stored sequence number value and increment duplicate_count
14:            if Duplicate_count > 3 then
15:                Reset duplicate_bit and duplicate_count to 0
16:            end if
17:            return true
18:        end if
19:    end if
20: end if
```

the IP address, and send an ARP reply to the requesting endpoint. The benefit of using time versus simply counting the number of unanswered ARPs to dynamically determine unused IP addresses is that it minimizes the chance of an adversary using this ARP reply behavior to fingerprint *Greasy*. An adversary could potentially observe that a certain number of unanswered ARP requests are followed by an ARP reply, which may seem suspicious. By using a configurable time interval, the chance of recognizing *Greasy*'s ARP behavior is minimized, which is why we opted to keep *LaBrea*'s time-based algorithm for determining unused IP addresses. If *Greasy* sees a gratuitous ARP request/reply for a captured IP address, *Greasy* will release the IP and place it in the *REAL_IP* list. A gratuitous ARP request/reply packet contains the source and destination IP address of the machine sending the packet, and broadcast MAC address ff:ff:ff:ff:ff:ff as the destination MAC address. It is primarily used to notify other machines in the LAN of updates to other machines' ARP tables. For instance one of the applications for a gratuitous ARP request/reply is to update the ARP tables of other local hosts so they know an IP interface or link has gone up; this

also notifies other hosts that a particular IP address is now in use, so as to prevent collisions in IP address usage [59]. *Greasy* uses the gratuitous ARP request/reply information to guarantee it does not continue to capture an IP address that is now occupied by a real host. *Greasy* also identifies live hosts by looking for ARP replies. In addition, *Greasy* will only capture the IP address if the IP address is part of the configured fraction of IP addresses that should respond in the subnet. In Section 3.2.1 we discussed how high occupancy subnets may be indicative of the existence of tarpit hosts. The method for determining whether an IP address should be captured based on the configured occupancy was discussed in Section 3.2.1. Algorithm 1 shows the ARP handler algorithm.

TCP_Handler

The TCP_Handler Class Module contains the heart of *Greasy*'s tarpitting functionality and is composed of the handler function and two helper functions:

- *void handle_TCP(TCP *tcp_recv, IP *ip_recv, EthernetII *eth_recv, RawPDU *raw_recv)*
- *bool build_TCP(TCP & myTCP, TCP *recvTCP, IP *recvIP, RawPDU *recvRaw)*
- *bool tarpit_opt(int switchNum, TCP *t_recv, TCP & my_t, TCP_tuple key, RawPDU *r_recv)*

The purpose for the TCP_Handler Class is to process all incoming TCP packets, and execute *Greasy*'s tarpitting functionality by utilizing heuristics for random tarpitting behavior, as discussed in Section 3.2.2. The following sections explain the roles of each function in the TCP_Handler Class.

void handle_TCP(...) : The purpose of this function is to check to make sure that the target IP address of the associated incoming TCP packet is either in the hard-captured or soft-captured IP lists. If so, it then checks to see if the target port of interest is open on the particular tarpit hostIP address (Section 3.2.1). The function then invokes the helper function *bool build_TCP(TCP &myTCP, TCP *recvTCP, IP *recvIP, RawPDU *recvRaw)* to construct a TCP packet and determine whether that TCP packet should be sent to the *GreaseMonkey* instance to be added to the *send_q* (Section 3.3.3). This function takes as parameters pointers to various layers of the incoming packet (i.e., TCP, IP, Ethernet, and

Raw payload) to extract required information to construct a TCP packet. The algorithm is depicted in Algorithm 2.

bool build_TCP(...) : This helper function (Algorithm 3) constructs a TCP packet in response to particular flags set in the incoming TCP packet, and randomly chooses one of the four tarpitting behaviors to execute. *Greasy* responds to TCP packets as follows:

- *Greasy* does not respond to TCP packets with Reset (RST), FIN, or FIN+ACK flags set. FIN or FIN+ACK packets are an indication that the remote end is attempting to end the established connection. Because we want *Greasy* to hold that connection for as long as possible, we ignore their packet, and instead allow the connection to end after the timeout.
- *Greasy* sends a SYN+ACK packet in response to an incoming SYN packet
- *Greasy* sends a RST packet to incoming SYN+ACK packets. *Greasy* does not send SYN packets to establish a TCP connection to any endpoint, so this is not a legitimate packet.
- *Greasy* may send an ACK packet in response to incoming ACK packets depending on the particular tarpit behavior chosen. The helper function *bool tarpit_opt(int switchNum, TCP *t_recv, TCP &my_t, TCP_tuple key, RawPDU *r_recv)* generates the particular tarpit mode behavior and finishes preparing TCP packets, if applicable, for delivery.

In addition, the function generates an Initial Sequence Number (ISN) to be used in the SYN+ACK packet (i.e., the second packet in a TCP handshake) that will also be used as a stateless session identifier to verify that incoming ACK packets have already established a TCP handshake for a particular session. This works in much the same way as the SYN cookie (Section 3.1.2). We generate the ISN by using a custom hash modified from this example [47] to create a deterministically random 32-bit value (see (3.3)). All subsequent ACK packets with an established TCP handshake should have an ACK value of the ISN + 1.[10] By using this method, *Greasy* can verify the legitimacy of TCP ACK packets without

[10]The ACK value of the third packet in a TCP handshake is the sequence number from the SYN+ACK packet incremented by 1. Because *Greasy* never sends any payload, any subsequent ACK packets received during the TCP session should have the value of the ISN + 1.

having to store this information.

$$((senderIP * 59) \oplus targetIP \oplus (senderPort << 16) \oplus targetPort \oplus 0x62EA5E) \quad (3.3)$$

One additional point to make is that state is maintained in order to execute the duplicate ACK tarpitting behavior, as discussed in Section 3.2.2. After verifying an ACK packet using the ISN *Greasy* adds an entry for the session using source and destination IP addresses and port numbers. The tuple value for the entry will be used by the helper function *bool tarpit_opt(...)* to determine how to process the duplicate ACK tarpitting behavior. This tuple contains information for the sequence number of the last ACK packet received, the duplicate_bit, and the duplicate_count value. If the duplicate_bit is set to 1, then this *build_TCP(...)* function will choose the duplicate ACK tarpitting behavior in lieu of choosing a tarpit mode randomly. Otherwise if this is 0, then this function will choose a tarpitting mode at random.

The function *build_TCP(...)* takes as parameters pointers to various layers of the incoming packet (i.e., TCP, IP, and Raw payload) to extract required information for the construction of the TCP packet, and a reference to the TCP packet constructed in the *handle_TCP(...)* function. *build_TCP(...)* returns either true or false to tell *handle_TCP(...)* whether to send the constructed TCP packet. The algorithm for this function is depicted in Algorithm 3.

bool tarpit_opt(...) : The purpose of this helper function is to generate the specific tarpitting behavior as chosen by the *build_TCP(...)* function, and returning either true or false to the *build_TCP(...)* function, and subsequently the *handle_TCP(...)* function to indicate whether or not to send the constructed TCP packet to the *GreaseMonkey* instance. The four tarpitting behaviors, implemented as case statements, are:

- Partial ACK mode
- Persistent mode
- Non-persistent mode
- Duplicate ACK mode

The partial ACK mode is generated by taking the size of the raw payload of the incoming ACK packet and dividing that size using a denominator value chosen at random using

rand() from a static set of values (i.e., ranges from 3 to 8). The resultant value is then added to the incoming ACK packet's sequence number and stored as the ACK value of *Greasy*'s constructed TCP packet. It returns true.

The persist mode is generated by setting the receive window of the constructed TCP packet to zero and returns true. The non-persistent mode is achieved by not responding to the incoming ACK packet and returning false.

The duplicate ACK mode first checks the duplicate_bit stored in the tuple value of the TCP session entry. If the bit is 0, then *Greasy* sets the bit to 1, drops the packet, and returns false, forcing the remote end to retransmit the packet. The purpose for dropping the packet at this step is just to encourage more retransmissions from the remote end. *Greasy* begins sending duplicateACK packets the next time it receives an ACK packet. If the bit is set to 1, then *Greasy* checks to make sure the stored sequence number for the TCP session is less than or equal to the sequence number of current incoming ACK packet to ensure *Greasy* does not ACK a value larger than the current sequence number. If it is, then *Greasy* sets the ACK value for the constructed TCP packet using the stored sequence number, increments the duplicate_count value by 1, and returns true. This mode will send a duplicate ACK packet two additional times for a total of 3 duplicate ACK packets to mimic the behavior of out-of-order packets sent by the remote end. After the third duplicate ACK packet is sent, *Greasy* sets the duplicate_bit and duplicate_count values to 0.

The *tarpit_opt(...)* takes as parameters the case statement number that indicates the tarpitting mode chosen, pointers to the TCP and Raw payload layers of the incoming packet, the TCP session entry key to access the tuple data, and a reference to *Greasy*'s constructed TCP packet. The algorithm is depicted here Algorithm 4.

ICMP_Handler

The *ICMP_Handler* impersonates a real host by sending ICMP [60] echo replies to remote hosts. The handler first checks to see if the target IP of the incoming ICMP request is captured. If it is, then the handler will send an echo reply. The goal for handling ICMP echo requests is to create the illusion of live hosts, as adversaries typically use ICMP pings to quickly establish a list of active hosts on a network during their information gathering

phase. There is no tarpit signature associated with ICMP packet handling, and we do not expect to see *Greasy* perform any differently than *LaBrea* since *Greasy* uses *LaBrea*'s algorithm for handling ICMP packets.

3.3.3 Multi-threaded Application Design

The multi-threaded aspect of *Greasy* was designed and implemented by Professor Justin Rohrer. The purpose for including a multi-threaded component to *Greasy* is two-fold. The first objective is to parallelize tasks in order to handle the potentially large capacity of incoming network traffic to produce a highly scalable application. The second objective is to further modularize *Greasy*'s design by partitioning the different processing concerns into separate tasks. *Greasy*'s multi-threaded design consists of five or more threads, as seen in Figure 3.2. *Greasy* contains a *main* thread, whose primary responsibility is to perform all of *Greasy*'s program initializations as well as wait for other threads to finish in order to exit safely and cleanly. *Greasy* contains two other threads: *sniff_thread* and *fs_thread*. The purpose for the *sniff_thread* is to execute the sniffer in order to monitor the network, filter traffic by packet type, and place the packets onto the mutex protected priority queue called *pdu_q*. The *fs_thread*, also called *frame_send thread*, takes packets off the *PriorityQueue*, *send_q*, and send those packets off to the network. Each *GreaseMonkey* worker instance has a thread whose job is to remove a packet from *pdu_q*, process the packet using one of the three packet handlers discussed in Section 3.3.1, and place the packet to send in response to the processed packet (if applicable) onto *send_q*. There is also a thread for *Easylogging++*, a library package used for logging that is further discussed in Section 3.4. This thread is used by all the other classes for logging statements.

3.3.4 Configuration File and Parser

Greasy includes a configuration file that provides the user the flexibility to choose initialization and operational settings. The configuration file is partitioned into three main areas: initialization, IP capture settings, and tarpit behavior settings. Initialization settings include:

- Specification of the device interface for *Greasy*'s sniffer
- Manual specification of the IP address and subnet for a non-configured interface

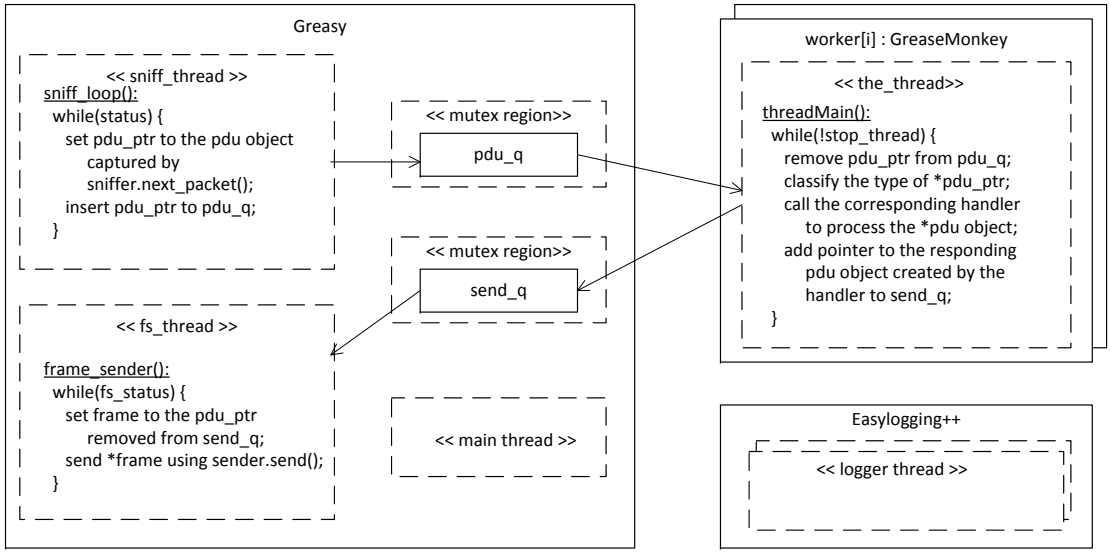

Figure 3.2: Multi-Threaded Design. The arrows in the figure show the control flow of tasks among the threads and queues. *Greasy* consists of five or more threads.

- And the option to ignore an initial ARP used before the start of the program in order to determine live hosts and exclude them from being tarpitted.

The IP capture settings include:

- Indication of a switched environment[11]
- Specification of IP addresses to hard-capture and the option of specifying a particular fraction of the subnet to capture
- Specification of IP addresses to exclude from capture
- Setting the soft capture option and choosing a particular fraction of the subnet to capture
- Setting the maximum and minimum values for the ARP handler as explained in Section 3.3.2

[11] A switched environment prevents *Greasy* from seeing potential ARP replies in response to ARP requests because the packet frame is sent directly to the port of the requesting host instead of being broadcast to all ports on a device.

The following configuration settings are available for tarpit behavior:

- Setting the maximum receive window size as discussed in Section 3.2.2
- Specifying *Greasy* listen to all ports. This setting is available but highly discouraged as described in Section 3.2.1.

3.4 Implementation

Greasy is written in native C++ and uses STLs. A large bulk of the program calls classes and functions from the *Libtins* packet crafting library [61]. The *Libtins* classes used by *Greasy* include: *AddressRange, ARP, EthernetII, IP, ICMP, NetworkInterface, PacketSender, RawPDU, Sniffer, SnifferConfiguration, TCP*, and *Timestamp*. We chose to use this particular packet crafting library because it is C++ compatible, IPv4 and IPv6 enabled, and cross-platform, which is useful for when we extend *Greasy* to be used on Windows and other operating system platforms. We also used *Libconfig* [62] for configuration management, *Easylogging++* [63] for its light-weight, robust, and performance logging capabilities for C++11 or higher, and *POSIX* threads (also known as *pthreads*) [64] for standardized C++ language threading programming interface. The development platform was a 64-bit *Ubuntu 14.04* virtual machine.

CHAPTER 4:
Experiments and Results

We conducted several experiments using a select set of metrics to measure the impact of implementing new tarpitting capabilities and other improvements in *Greasy*, particularly in the areas of deception and tarpitting performance. This chapter describes the chosen metrics, experimental set-up and design, and analysis of the results.

4.1 Metrics

We chose three metrics to evaluate *Greasy*'s deceptiveness and tarpitting ability compared to that of an existing tarpit application, *LaBrea*. The metrics included are:

- Effectiveness Against *Degreaser*
- Degree of Stickiness
- Packet Latency

Other metrics examined but not used in our evaluation of *Greasy*'s improvements include CPU utilization, memory consumption, and reliability. These metrics are useful in evaluating overall tool performance, but are out of the scope of this thesis, since we focus our attention on measuring one particular aspect of *Greasy*'s performance (i.e., tarpitting performance). Our chosen metrics are directed at measuring *Greasy*'s tarpitting improvements.

4.1.1 Effectiveness Against *Degreaser*

One of the main objectives for this thesis is to better obscure network tarpits from tarpit scanners like *Degreaser*. This metric measures the number of tarpit hosts versus real hosts identified in a given subnet. *Degreaser* scans a subnet or list of subnets and classifies each responding IP address as unresponsive, a tarpit, or a real host. Depending on the particular tarpit-like behavior observed from incoming TCP SYN+ACK and ACK packets, *Degreaser* further classifies the IP as a particular type of tarpit (e.g., *LaBrea* persistent or non-persistent, *iptables* tarpit or *iptables* delude, or unknown) [5]. We use *LaBrea* as our

control, and run *Degreaser* on both *LaBrea* and *Greasy* tarpit hosts. We expect *Degreaser* to classify the large majority of *Greasy* tarpit hosts as real hosts, since we mitigate tarpit signatures (i.e., small TCP window size and lack of TCP options) that *Degreaser* uses to identify tarpits.

4.1.2 Degree of stickiness

We also want to measure the effectiveness of *Greasy*'s tarpitting capability (i.e., its stickiness, or ability to hold onto a connection for longer periods of time). We further break this category into two sub-metrics:

1. Number of Packets Transferred
2. Duration of Tarpit Connections

Number of Packets Generated

We claim that the number of packets transferred during a tarpitted connection between a tarpit and remote host within a given timeframe may be indicative of the tarpit's ability to keep a remote host engaged and the connection open (i.e., an indication of the stickiness of the connection). For instance, a remote end engaged in a connection with a tarpit host may end up expending time and bandwidth resources to send large amounts of packet retransmissions or zero window probes to hold the connection open. However, there is also the possibility that the remote end may consume resources, like memory, to hold a connection open without sending or receiving traffic. We use this metric to help us correlate the number of packets transmitted in a tarpitted connection and *Greasy*'s ability to hold a sticky connection.

We measure this metric by calculating the number of packets both sent and received by a tarpit in a given connection. We define traffic going from one endpoint to another endpoint in a connection as a flow identified by a 5-tuple identifier: source IP address, destination IP address, source port number, destination number, and protocol. For instance, traffic from host *A* to host *B* is considered a flow with a 5-tuple identifier, and traffic from host *B* to host *A* is considered another flow with a different 5-tuple identifier. We infer which two endpoints are associated in a particular tarpitted connection by their respective 5-tuple identifiers. A flow may expire if it receives a FIN+ACK packet, RST packet, or sees a SYN

or SYN+ACK for the establishment of a new TCP handshake [65]. We restart the counters for the number of packets sent and received by a tarpit when a TCP SYN+ACK packet is observed in a tarpit flow, indicating the start of a new tarpit connection between the tarpit and the same remote end.

Duration per Connection

The length of time, in seconds, that a tarpitted connection remains open is also indicative of the stickiness of the tarpit. Our intuition is that the longer the tarpit can hold a connection with the remote host, the more effective a tarpit is at exhausting the remote host's time, bandwidth, and memory in keeping the connection open, as well as delay the remote host from scanning elsewhere. We measure this metric by using packet timestamps found in a packet capture to calculate the duration for a given tarpitted connection. Using the 5-tuple identifier, we infer the two endpoints associated with a particular tarpitted connection and calculate the total time of the connection. The counter for the duration of a given tarpitted connection restarts when a TCP SYN+ACK packet is observed in the flow from the tarpit.

4.1.3 Packet Latency

We also want to measure performance advantages or disadvantages of *Greasy* compared to *LaBrea* using our packet latency metric. Specifically, this metric measures the amount of overhead in packet handling (if any) *Greasy* incurs as observed through one aspect of end-to-end latency. We claim that *Greasy* may be more detectable if an adversary observes a noticeable packet latency. End-to-end latency is a function of transmission, propagation, queuing and processing delays [66]. Thus, we may see a wide variety of observed Round Trip Time (RTT) results on the Internet from any given vantage point. Transmission propagation and queuing delays are affected by packet length, link throughput, and link distance, among other factors, and may be constant values. Processing delay is implementation specific, and in our case, a potentially variable value. Our goal is to obtain a baseline RTT measurement for a particular network, and observe whether RTT values deviate from the expected RTT range. Deviation may be attributable to extra processing delay by the tarpit application, or perhaps another latency factor, such as queuing delay. We do expect that RTT measurements between two IP addresses on the same subnetwork will not have

a statistical difference between their delay distributions because latencies caused by propagation and transmission delays, for example, should be minimized due to a shorter link distance. Thus we include a vantage point on the subnetwork to mitigate or minimize the effects of some end-to-end latencies.

We use a modified version of Trabelsi *et al.*'s RTT detection technique to measure this metric [67]. In their work, Trabelsi *et al.* use this technique for the detection of sniffers on the LAN. Using the RTT of ICMP [68][12] packets and a statistical model (z-statistic [69]) and hypothesis testing they were able to distinguish between hosts in normal mode and those in promiscuous mode [70] (i.e., sniffing mode). This technique is used for our packet latency metric because both *LaBrea* and *Greasy* use sniffers that monitor the network promiscuously to capture packets. We first gather training data (i.e., RTT measurements from ICMP packets sent to a control subnet with real hosts), then gather RTT measurements from both *Greasy* and *LaBrea* subnets. Using this data we compute RTT averages, standard deviations, and percent of change from the training data. To determine whether the RTT samples of *Greasy* and *LaBrea* are distinguishable from that of the control, we use hypothesis testing. Our null hypothesis is that the tarpit sample and control population are the same population, and our alternative hypothesis is that they represent different populations. We then calculate the z-statistics for both *Greasy* and *LaBrea* using (4.1).

$$z = \frac{M - \mu}{SE} \tag{4.1}$$

M is the sample tarpit mean, μ is the control mean, and SE is the standard error unit. This z-value indicates the number of standard deviations the average RTT value of the tarpit is from the average RTT value of the control. SE is defined in (4.2):

$$SE = \frac{\sigma}{\sqrt{n}} \tag{4.2}$$

[12]ICMP offers a system-level feedback communication mechanism independent of the host, and as such ICMP delays are generally decoupled from a host's processing load. In fact, many ICMP packets are directly processed and sent on the Network Interface Card (NIC) [68]. ICMP packets make excellent candidates for packet latency measurements because their processing delays are typically independent of the host's load.

where σ is the control standard deviation, and n is the sample tarpit count. We use a two-tailed test [69] to capture extreme z-values, and use the critical values 2.58 and -2.58 to give us a 99% confidence level whether to reject or accept the null hypothesis. If the calculated z-statistic is greater than or equal to 2.58 or less than or equal to -2.58, we can then reject the null hypothesis with 99% confidence. We expect *Greasy* to have some packet latency due to packet handling functions and implementation of the *Libtins* sniffer. The *Libtins* sniffer is implemented in kernel-space, and handles all the input and output actions for packets captured or sent by the sniffer. The sniffer may experience delays due to kernel processing or the underlying mechanism the *Libtins* sniffer may interact with the kernel. The rest of the packet parsing/processing and packet crafting utilities for *Greasy* are implemented in user space, which may also be the source of packet latency due to specific function implementations, as well as delays due to transferring packets and control between kernel and user space.

4.2 Experimental Setup

4.2.1 Platform

We deployed *Greasy* and *LaBrea* on a *FreeBSD* 10.2-RELEASE-p7 i386 platform located on an East Coast US network with access to a /17 network telescope [71].[13]

Figure 4.1 shows the set-up of both *Greasy* and *LaBrea* tarpit applications on the Internet-facing network telescope. Each tarpit application occupies a /24 subnet on the telescope's /17 subnet. In the figure, the arrow shows traffic coming into the network telescope from the Internet. The elipses on either side of the /24 subnets indicate other /24 subnets that are on the /17 subnet. We wanted to deploy *Greasy* and *LaBrea* on two adjacent /24 subnets, however the amount of incoming traffic to each /24 subnet was a factor we could not control. In order to decrease the amount of variability in our results due to traffic load variance, we monitored network traffic on the /17 IP space using *tcpdump* [72], a command-line packet analyzer, for one month in October 2015 in order to determine which two adjacent /24 subnets received similar amounts of traffic. For the most part, the majority of /24 subnets

[13]Network telescopes utilize unused IP address space for monitoring activity on the Internet at large. For instance, they can passively collect incoming traffic, or "backscatter" from a Distributed Denial of Service (DDoS) attack [71].

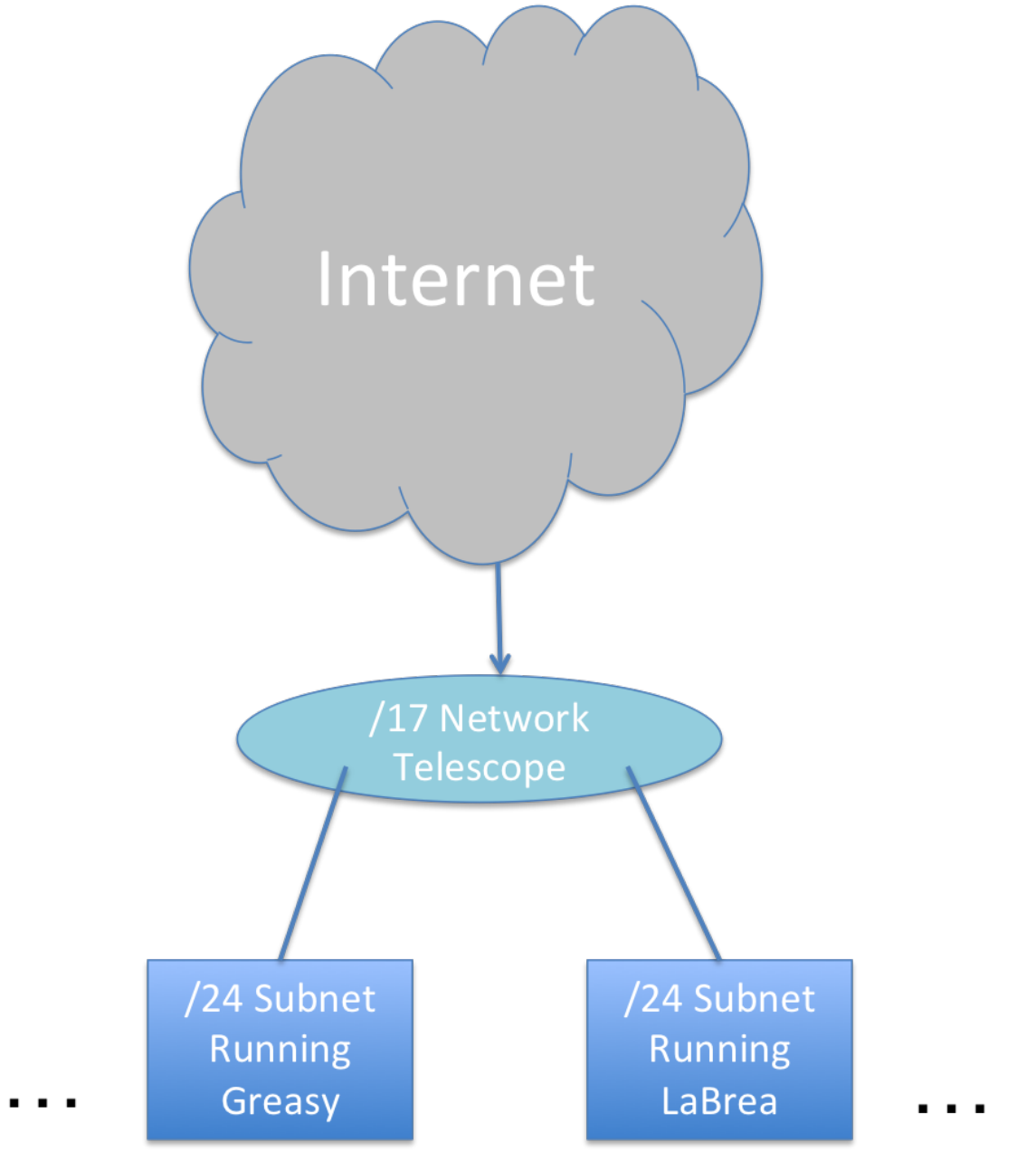

Figure 4.1: Experimental Setup for *Greasy* and *LaBrea*/24 Subnets on the /17 Network Telescope

received around 200,000 packets daily, or approximately 2.3 packets per second. Figure 4.2 is a snapshot of one day's worth of incoming network traffic that was seen in our daily measurements throughout October. There was one subnet in particular that was an

outlier for all packet captures, receiving as many as 800,000 packets per day (9.3 packets per second). This result is attributed to large amounts of ICMP ping scans and TCP SYN scans to one particular IP address. Other subnets also experienced scanning from the same IP addresses but no where near as large a scanning volume. Since the majority of the subnets received similar amounts of traffic, we arbitrarily chose to run *Greasy* and *LaBrea* on two adjacent anonymized /24 subnets.

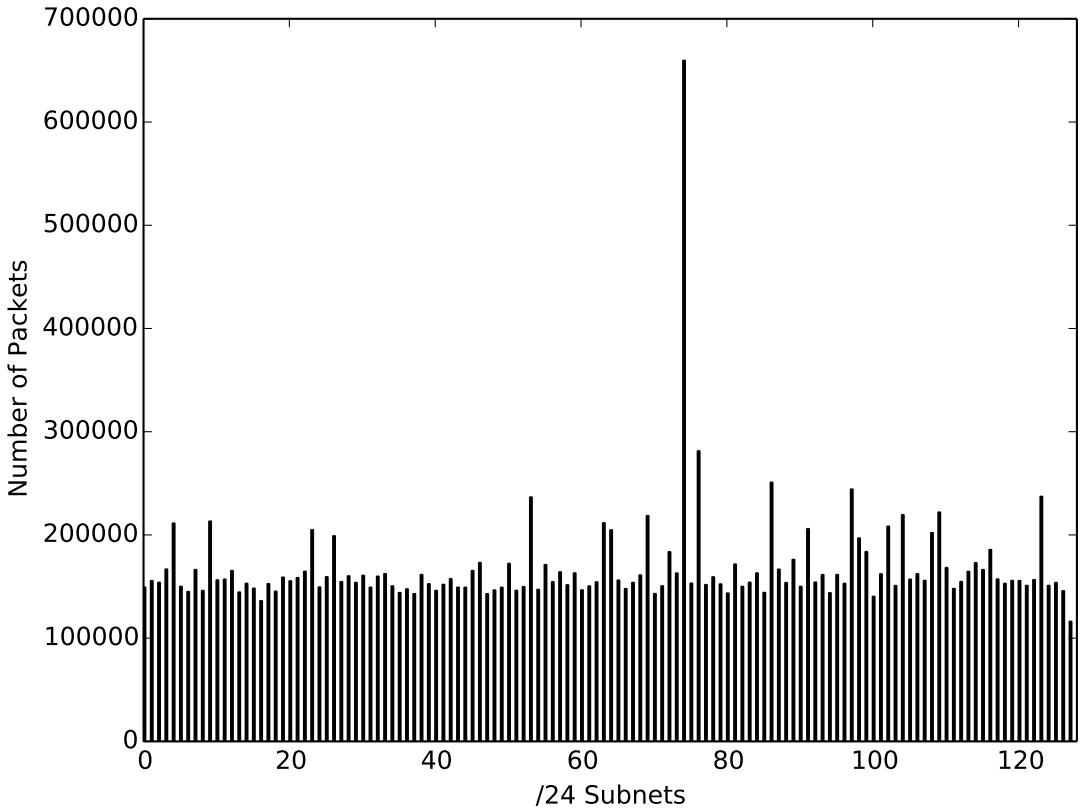

Figure 4.2: Number of Packets Received Per /24 Subnet of the /17 Subnet on 10/27/15. This is just a snapshot of daily traffic seen in October.

4.2.2 Procedures

Degreaser Experiment

We ran *Degreaser* on a *CentOS Linux release 7.2.1511* machine located at Naval Postgraduate School (NPS). The *Degreaser* program takes as inputs the target subnets and network device interface used to capture packets, and outputs results as specified in Section 4.1.1.

Stickiness Experiment

To measure the stickiness of the tarpits, we conducted the following procedures for the experiments:

- Configured *Greasy* to capture 100% of the IP addresses in its /24 subnet. (*LaBrea* captures 100% of the IP addresses by default).
- Configured both *LaBrea* and *Greasy* to hard-capture IP addresses, rather than capture IPs dynamically in order to create a more controlled experiment environment.
- Ran *Greasy* and *LaBrea* on their respective /24 subnets for 28 days total:
  - 14 days with *LaBrea* configured to tarpit connections in non-persistent mode
  - 14 days with *LaBrea* configured to tarpit in persistent mode
- Collected network traffic using the *tcpdump* utility, and filtered packet capture files to isolate packets from *Greasy* and *LaBrea* hosts.
- Parsed packet capture files to obtain measurement parameters as noted in Section 4.1.2, and calculated *Greasy*'s degree of stickiness compared to that of *LaBrea*.

In addition, because both /24 subnets may receive variable amounts of traffic, we ran *Greasy*, *LaBrea* in persistent mode, and *LaBrea* in non-persistent mode each for 24-hours and collected network traffic using *tcpdump* to obtain data for our degree of stickiness experiments.

We also checked for possible outages by making sure that the programs were still running on the *FreeBSD* machine everyday and that there was traffic coming from the two /24 subnetworks. We checked the latter by reviewing the packet capture files. There were no outages observed for the 28 days.

Packet Latency Experiment

Packet RTTs may be altered by distance (i.e., the number of intermediate devices the packet must travel through to get from point A to point B) and network congestion, among other factors. The machine running both *Greasy* and *LaBrea* is located in Massachusetts, whereas we are located in California, so our RTT results may be skewed by distance and network interference. Our control /24 anonymized subnet is also located in Massachusetts. To get a more accurate picture of packet latency, we used three different vantage points to ping *Greasy* and *LaBrea* hosts: a *Mac OS X Version 10.10.5* machine on a home residential network in California, the *CentOS Linux release 7.2.1511* machine at NPS, and the *FreeBSD 10.2-RELEASE-p7* machine on the telescope. We sent three pings to each host in each subnet (i.e., subnets with *LaBrea*, *Greasy*, and real hosts), and calculated the average RTTs, standard deviations, percent of change compared to our control sample, and z-values for both *Greasy* and *LaBrea*. 254 hosts were up on both *Greasy*'s and *LaBrea*'s subnets and only 14 hosts were up on the control subnet. We sent a total of 762 pings to *LaBrea* and *Greasy*'s subnets, and 42 pings to the control subnet. We did not experience any packet loss when pinging from the local East Coast vantage point, which was expected since it was on the same subnetwork as the three subnets. We did experience some packet loss from both our NPS and residential vantage points, around 3-5 packets out of 762 pings to each subnet, but that did not affect our overall results because of the large amount of RTT measurement data we obtained and because we ran this experiment three times to verify our results.

4.3 Results and Analysis

The following section details the results of our experiments, and analyses of *Greasy*'s deceptiveness, degree of stickiness, and measure of packet latency overhead.

4.3.1 Experiment: Deceptiveness

As shown in Table 4.1 *Degreaser* classified 100% of *LaBrea*'s hosts as tarpits, whereas *Degreaser* was only able to classify 2.8% of *Greasy*'s hosts as tarpits. *Degreaser* classified all hosts without TCP options and/or a window size smaller than 20 bytes as a tarpit. The 2.8% of *Greasy*'s hosts were identified as *LaBrea* tarpits because they exhibited window sizes of 0 bytes when they executed the persistent mode tarpit behavior. We cannot use

persist mode with a window greater than 0 bytes because persist mode is required to have a 0 window. One way to mitigate this detection might be to decrease the number of times persist mode is chosen for *Greasy*'s random tarpitting behavior, or eliminate this tarpitting behavior entirely. However, persist mode proves to be quite effective at tarpitting a connection, as will be discussed in Section 4.3.2, so we definitely still want to keep persist mode as a tarpitting option. Despite the small percent of identified tarpits, our results show that *Greasy* successfully deceived *Degreaser* by implementing randomized tarpit behavior and adding TCP options. We expected this outcome because previously tarpit signatures were trivial to detect. However, in addition to mitigating these signatures, we further improve tarpitting features and raise the cost for adversaries to detect *Greasy*.

Table 4.1: Number of Real Hosts versus Tarpit Hosts Identified by *Degreaser*

Classification	*LaBrea*	*Greasy*
Number of Real Hosts	0	247
Number of Tarpit Hosts	254	7

4.3.2 Experiment: Degree of Stickiness

We collected two sets of data to measure *Greasy*'s degree of stickiness compared to *LaBrea*: 14 days of network traffic with *LaBrea* running in non-persist mode only and 14 days of network traffic with *LaBrea* running in persist mode only. *Greasy*'s configurations did not change throughout the 28 days. Because the speed of network scanning occurs on the order of seconds [73], we measured the degree of stickiness for both tarpits at the hour-level of granularity and binned all results into one-hour intervals. We also noticed that much of the network traffic observed from our packet captures are TCP half-open scans, in which the remote end sends a TCP SYN packet, *Greasy* responds with a TCP SYN+ACK packet, and the remote end does not send another ACK to complete the handshake. Because we wanted to make sure these scans did not skew our data analysis, we created two sets of data for both 14-day periods. One set is comprised of both TCP half open scan traffic and TCP traffic with an established handshake (which we subsequently refer to as *total flows* in following discussions), and the other set only has TCP traffic with an established handshake (which we refer to as *handshake flows*).

Using the 5-tuple identifier as a key in a dictionary, we keep count of the number of packets

sent by each flow. Everytime the tarpit sends a SYN+ACK packet to establish a new TCP handshake, we store the previous value for number of packets sent by the flow and restart the counter. We then plot these values as boxplots and Cumulative Distribution Function (CDF) plots to get the overall distribution of packets within each hour bin.

In general we noticed that values returned from the data set comprised of *total flows* consisted of mean and median values 50% less than those from the data set comprised of *handshake flows* only. We include the boxplots and CDF plots from our *total flows* data set as reference but only mention results that come from the data set with *handshake flows* in following discussions. Because our goal is to measure the stickiness of the tarpitted connection, and tarpitting heuristics only begin after the establishment of a TCP handshake, we want to make sure we tailor our analyses to the tarpitting portion of the connection, rather than refer to data skewed by incomplete TCP handshakes.

Figure 4.3 shows the boxplots for packets sent per flow for *Greasy* compared to *LaBrea* in non-persist mode during the first 14 days. The distribution of the number of packet sent by *Greasy* ranges from 1 to 22,261 packets. The median range among the hour bins is 1 to 6 packets, with a mean of 13.6 packets sent per flow. Most of *Greasy*'s values lie below 2,500 packets per flow. The major outlier is the the flow with 22,261 packets. This value may be attributed to the duplicate ACK tarpitting behavior that requires the tarpit to send 3 duplicate ACK packets to the remote end to force the remote end to perform a packet retransmission. *LaBrea* has a distribution and median range of 1 packet per flow across all hour bins, and a mean of 1 packet per flow. This is expected because *LaBrea* non-persist only sends a TCP SYN+ACK in response to an incoming SYN and drops all other packets. Figure 4.4 is a CDF plot of the same results.

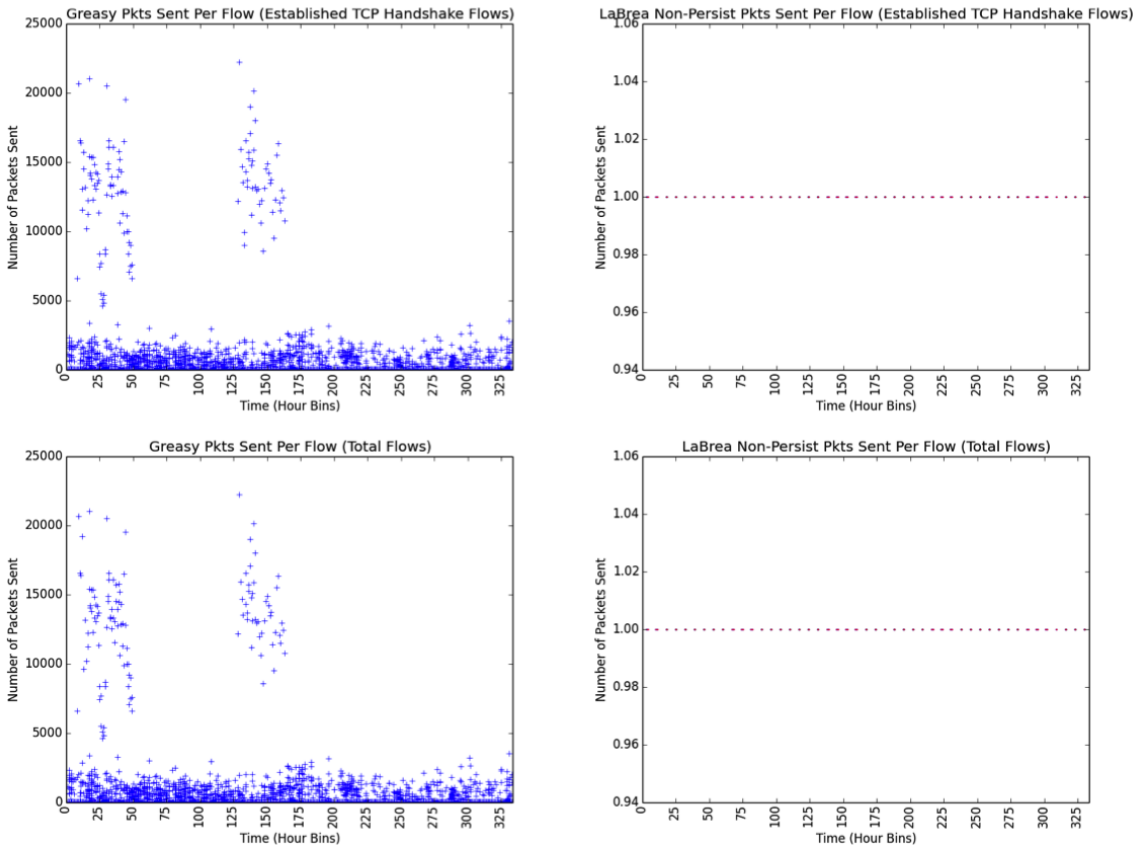

Figure 4.3: Boxplot of Packets Sent per Flow (*Greasy* versus *LaBrea* Non-Persist)

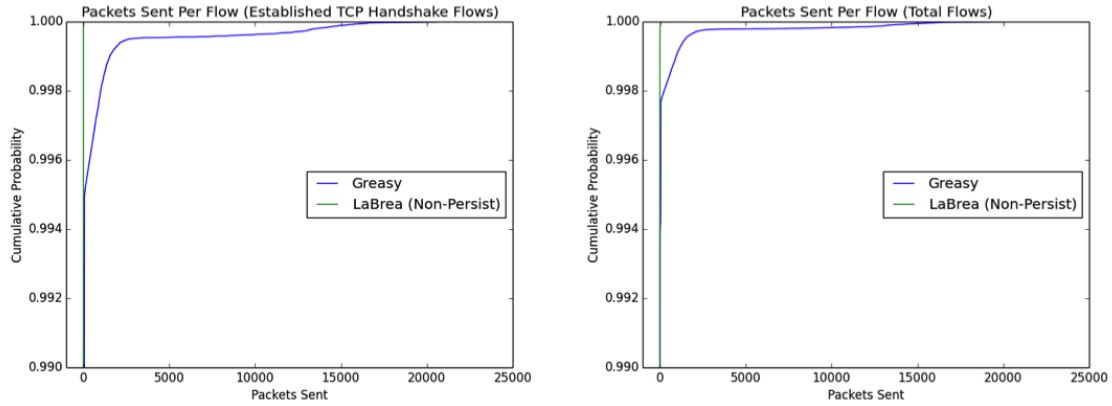

Figure 4.4: CDF of Packets Sent per Flow (*Greasy* versus *LaBrea* Non-Persist)

Figure 4.5 shows the packets received per flow for *Greasy* compared to *LaBrea* in non-persist mode. The distribution of packets received by *LaBrea* ranges from 1 to 51 packets per flow, while the median range among the hour bins is 2 to 10 packets per flow, and a mean of 2.7 packets per flow. The distribution for *Greasy*'s plots range from 1 to 22,261 packets per flow with a median range of 1 to 2 packets per flow and an average 12.6 packets per flows. Figure 4.6 represents the packet distribution as a CDF.

Based on our results, *LaBrea* in non-persist mode receives more packets than it sends, which is expected since *LaBrea* only sends 1 packet per flow. *Greasy*'s overall distribution of packets per flow, median values and mean values are similar for both packets sent and received by *Greasy*. This may indicate that *Greasy* sends as many packets as it receives to keep the remote end engaged in the tarpitted connection. The distribution range of packets received per flow for *Greasy* is larger than that for *LaBrea* in non-persist mode. *Greasy*'s plots have greater outliers and variability in results compared to *LaBrea* in non-persist mode. However, *Greasy*'s median ranges for packets received are slightly less than those for *LaBrea*. One explanation for this is that *Greasy* experienced many connections that end immediately after the TCP handshake is established, which is characteristic of some TCP SYN scans initiated by the remote end. The averages for the number of packets sent and received by *Greasy* are larger than those values for *LaBrea*, by about 300%.

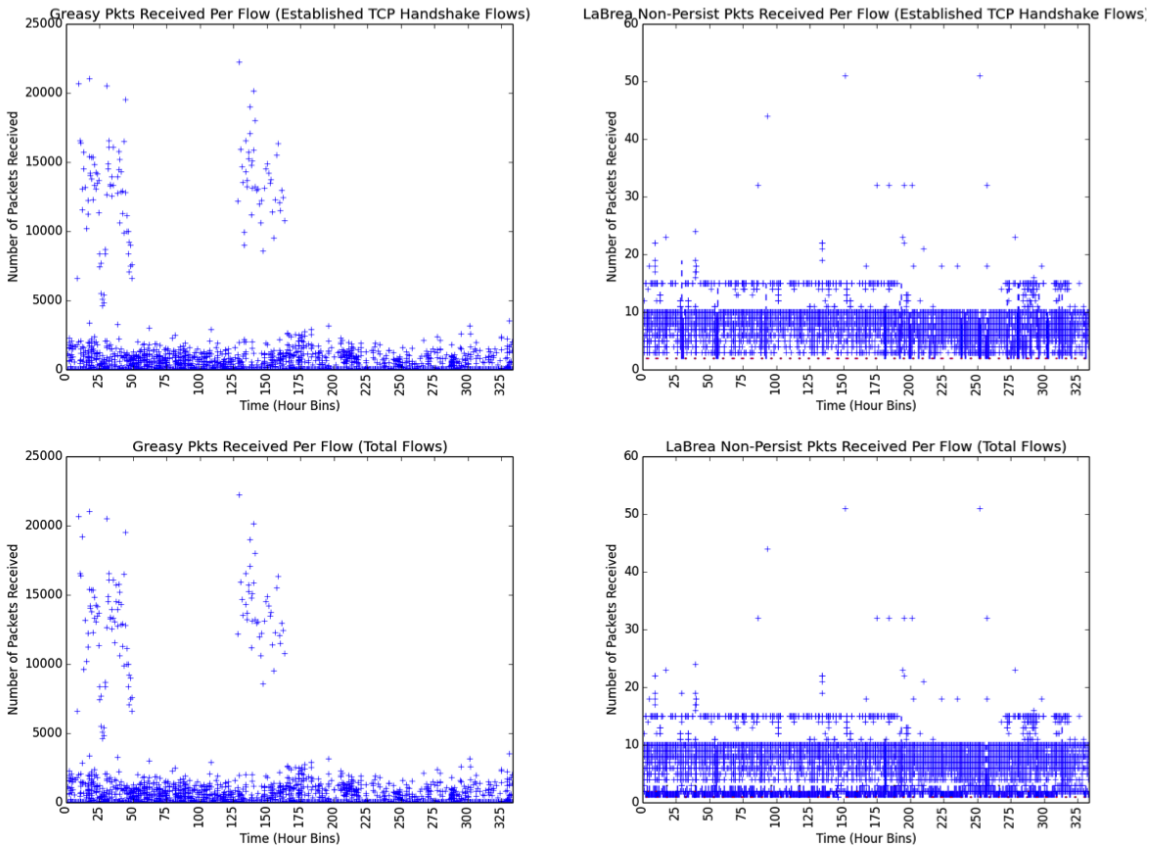

Figure 4.5: Boxplot of Packets Received per Flow (*Greasy* versus *LaBrea* Non-Persist)

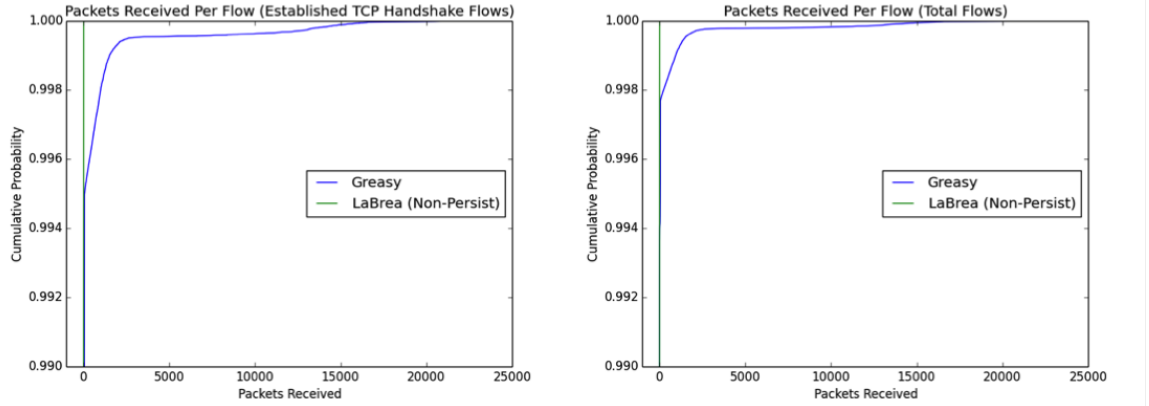

Figure 4.6: CDF of Packets Received per Flow (*Greasy* versus *LaBrea* Non-Persist)

Figure 4.7 shows packets sent per flow from *LaBrea* in persist mode versus *Greasy*. The distribution of packets sent for *LaBrea* ranges from 1 to 6,062 packets per flow, with a median range of 1 to 60 packets among the hour bins, and an average of 38.4 packets sent per flow. The distribution of packets for *Greasy* ranges from 1 to 4,572 packets per flow, with a median range of 1 to 7 packets among all hour bins and an average of 5.2 packets sent per flow. The CDF plot in Figure 4.8 for *total flows* shows that the plots for *LaBrea* in persist mode sends on average of 33.2 more packets in compared to *Greasy*.

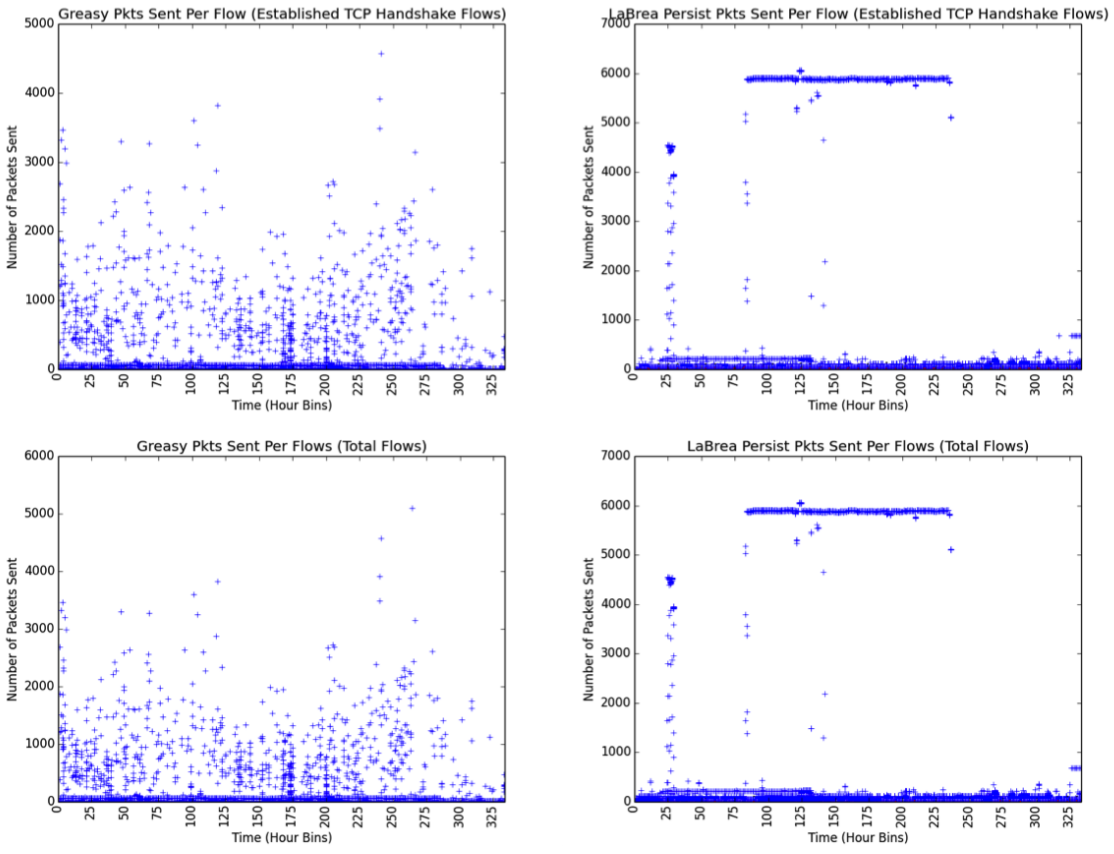

Figure 4.7: Boxplot of Packets Sent per Flow (*Greasy* versus *LaBrea* Persist)

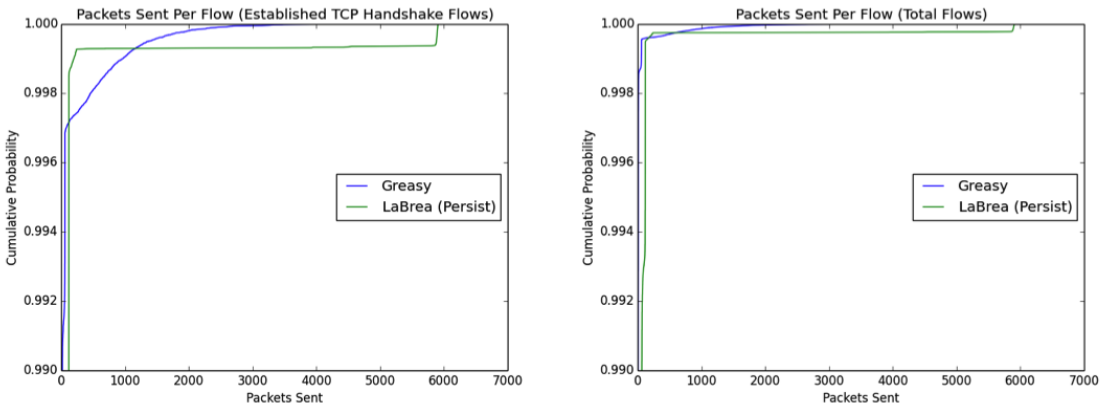

Figure 4.8: CDF of Packets Sent per Flow (*Greasy* versus *LaBrea* Persist)

Figure 4.9 and Figure 4.10 show the distribution of packets received by *Greasy* compared to *LaBrea* in persist mode. *LaBrea*'s packet distribution ranges from 1 to 6,062 packets per flow, with a median range of 2 to 60 packets per flow and an average of 38.4 packets per flow. *Greasy*'s packet distribution ranges from 1 to 4,565 packets per flow for *handshake flows* and 1 to 5,099 packets per flow for *total flows*. The median range *Greasy* is 1 to 2 packets per flow, with a mean value of 5.1 packets per flow.

The median ranges for packets received by *LaBrea* in persist mode are larger than those for *Greasy*. *LaBrea* has a cluster of values close to 6,000 packets across a number of hour bins for both plots for packets received and packets sent per flow. The plots show that *LaBrea* in persist mode sent and received on average 33 more packets than *Greasy*, which may be an indication that *LaBrea* holds a longer tarpit connection. It is important to note that the particular mean and median scores a completely dependent on the amount of traffic coming into subnetworks. During the first 14 days, *Greasy* had an average of 12 to 13 packets sent and received per flow. However, during the latter 14 days, *Greasy* only had an average of around 5 packets sent and received per flow. So although *LaBrea* outperformed *Greasy* by an average of 33 packets per flow, or 6 times that of *Greasy*, this magnitude of difference is not fixed. *LaBrea* in persist mode may have more packets transmitted per flow due to a longer connection duration.

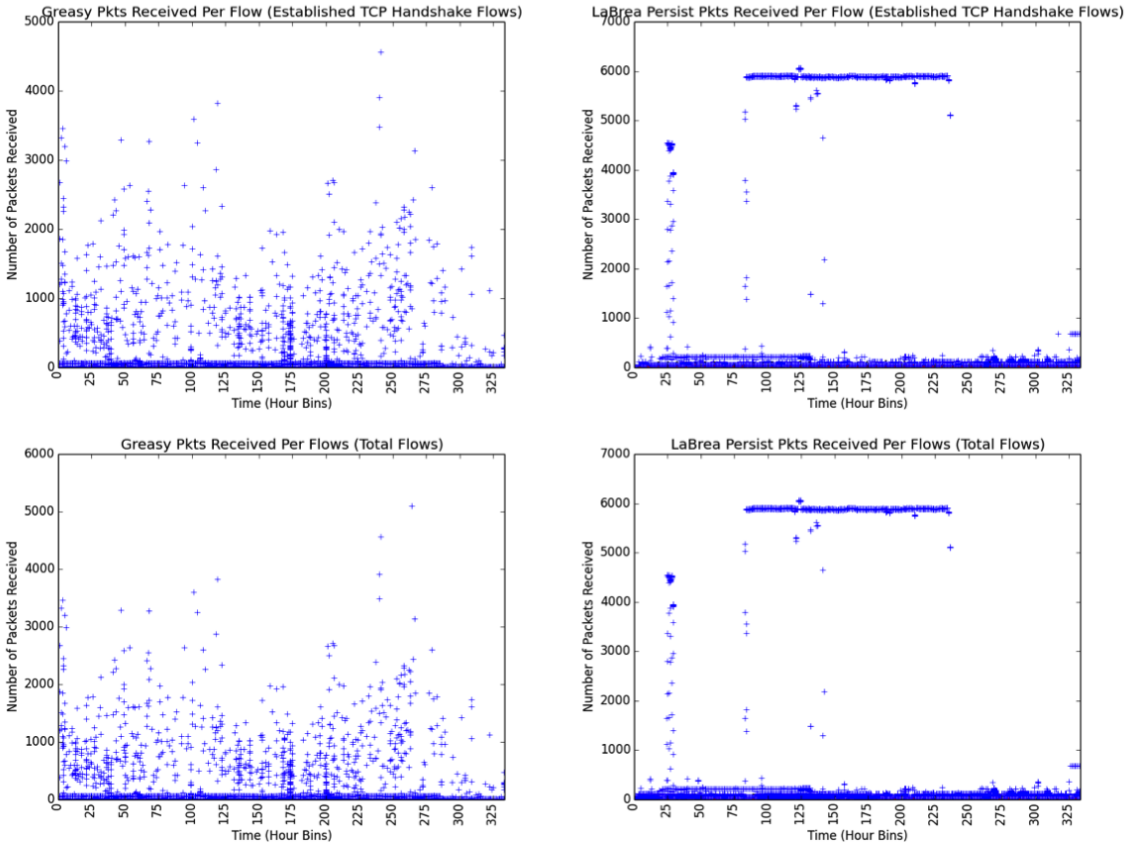

Figure 4.9: Boxplot of Packets Received per Flow (*Greasy* versus *LaBrea* Persist)

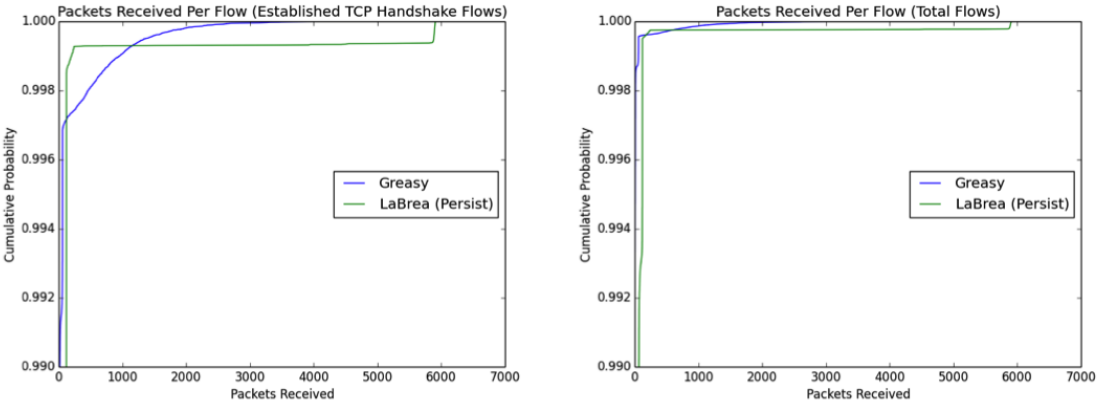

Figure 4.10: CDF of Packets Received per Flow (*Greasy* versus *LaBrea* Persist)

Next we analyze the duration per connection by using the 5-tuple identifier and identifying associated endpoints involved in a tarpitted connection to compute the duration per connection. Everytime the tarpit sends a SYN+ACK packet to establish a new TCP handshake, we store the previous value for the number of seconds per connection and restart the counter. We then plot these values as boxplots and CDF plots to get the overall distribution of duration values within each hour-bin. Similar to the results from number of packets sent and received per flow, we observe that data sets with *total flows* do tend to skew the data analyses by lowering median and mean values, compared to the data set composed only of flows from connections with TCP established handshakes.

Greasy's duration distribution ranges from 0 to 3,600 seconds per connection. The median range is 0 to 99 seconds per connection, with an average of 61.3 seconds per connection. The duration distribution for *LaBrea* in non-persist mode ranges from 0 to 2,056 seconds per connection, with a median range of 0 to 117 seconds per connection, and a mean of 7.1 seconds per connection. Although the maximum median value for *LaBrea* in non-persist mode is approximately 18 seconds greater than *Greasy*'s mean values, *Greasy*'s average duration is around 54 seconds greater than *LaBrea*'s average duration. Because *LaBrea* in non-persist mode only sends one packet (i.e., the second packet in the TCP conversation) the duration per connection is dependent on the amount of time spaced between retransmission packets and the quantity of retransmitted packets. Figure 4.11 shows that the majority of *LaBrea*'s duration outliers are approximately 150 seconds per connection and below, with sparsely scattered values in the 2,000 seconds range. *Greasy*, on the other hand, has duration outliers around 250 seconds per connection as well as a large number of duration values in the 3,500 seconds range. Figure 4.12 shows these results as a CDF plot. *Greasy* has a longer duration per connection compared to *LaBrea* in non-persist mode because *LaBrea* drops all packets after the initial TCP handshake, so the connection eventually times out on the remote end after a number of retransmission attempts. We expect *Greasy* to have a longer duration per connection because of its random tarpitting behavior that does not simply drop packets.

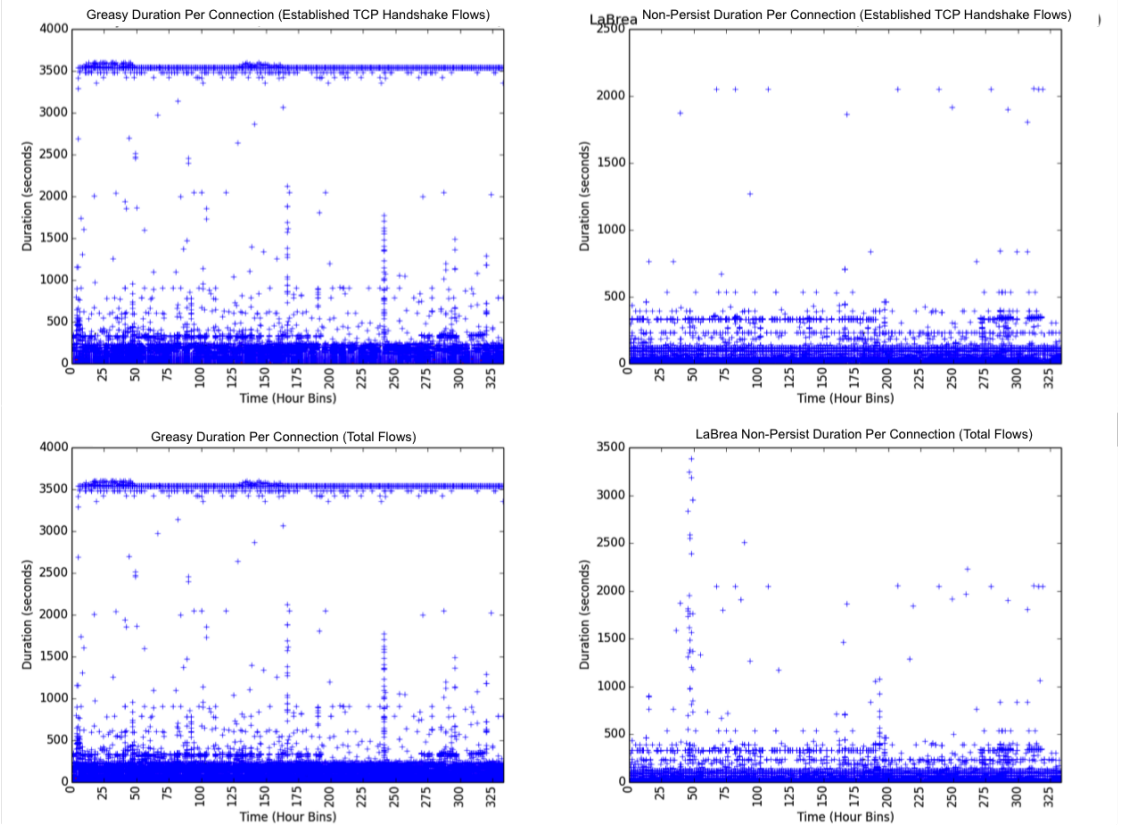

Figure 4.11: Boxplot of Duration per Connection (*Greasy* versus *LaBrea* Non-Persist)

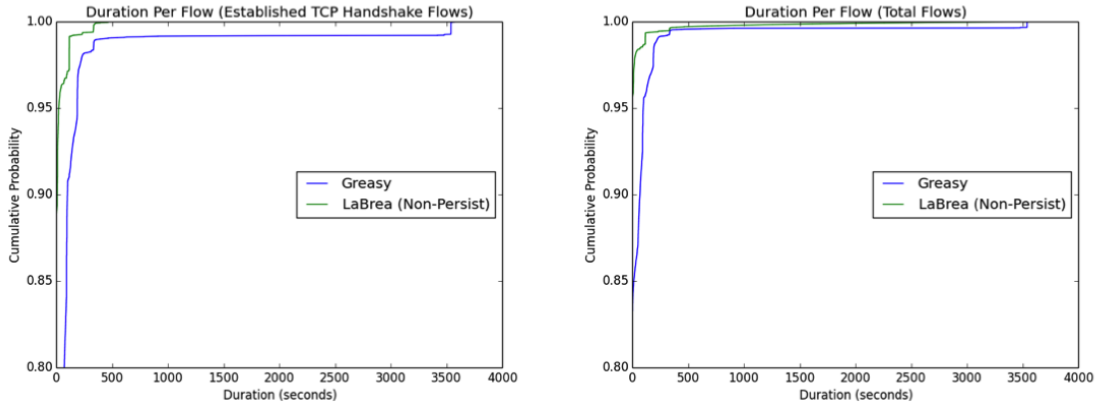

Figure 4.12: CDF of Duration per Connection (*Greasy* versus *LaBrea* Non-Persist)

Figure 4.13 shows the boxplot for the duration distribution for *Greasy* compared to *LaBrea*

in persist mode. *Greasy*'s duration distribution ranges from 0 to 3,562 seconds per connection. *Greasy*'s median range spans from 0 to 93 seconds, with an average of 45.7 seconds per connection. *LaBrea*'s distribution of values ranges from 0 to 3,600 seconds per connection. The median range of values spans from 0 to 3,535 seconds per connection, with an average of 2,316.0 seconds per connection. *LaBrea*'s tarpitted connections last on average almost 2,300 seconds longer than those of *Greasy*. Figure 4.14 further highlights the distribution of duration values from *Greasy* compared to *LaBrea* in persist mode. *LaBrea*'s duration results are significantly greater than *Greasy*'s, which were expected because of the nature of a persistent connection. As discussed in Chapter 2, the length of a persistent connection is dependent on the configurations of the remote end, so it is possible that a persistent tarpit connection can last indefinitely.

Because the volume and types of traffic hitting each subnet may vary, we chose a subnet to run *Greasy* for 24 hours, *LaBrea* in persist mode for 24 hours and *LaBrea* in non-persist mode for 24-hours. We re-ran the experiments that measure the tarpit's degree of stickiness and compare the results to those discussed earlier. We arbitrarily chose to use *Greasy*'s subnet to run each of the three tarpit applications during their 24-hour time-slot.

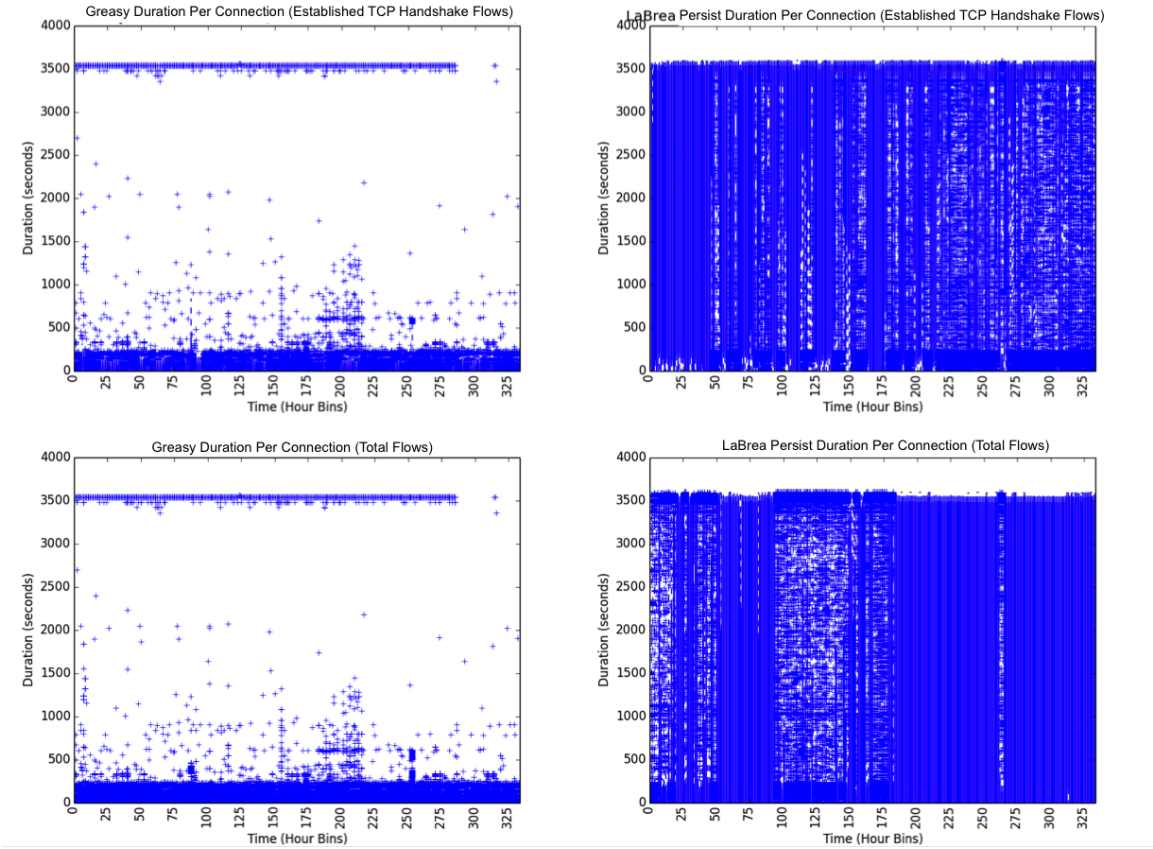

Figure 4.13: Boxplot of Duration per Connection (*Greasy* versus *LaBrea* Persist)

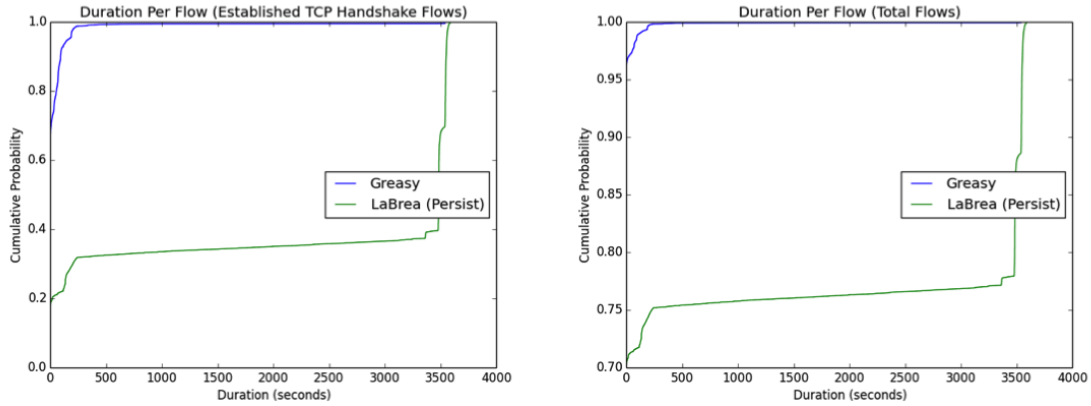

Figure 4.14: CDF of Duration per Connection (*Greasy* versus *LaBrea* Persist)

Figure 4.15 and Figure 4.16 show the duration distributions for *Greasy*, *LaBrea* in persist mode, and *LaBrea* in non-persist mode. We observe that *LaBrea* in persist mode had the greatest average duration per connection of 1035.7 seconds. *Greasy* averaged 18.1 seconds per connection, followed by *LaBrea* in non-persist mode that averaged 15.9 seconds per connection. These results are in line with our expectations, which indicates that the particular subnet on which the tarpit application ran did not affect duration results.

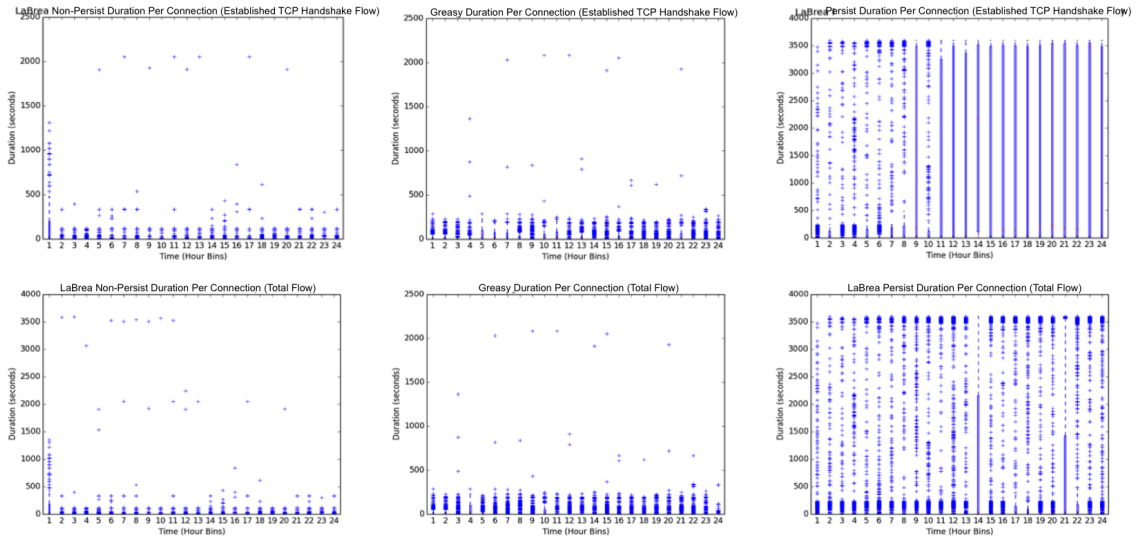

Figure 4.15: Boxplot of Duration per Connection (Same /24 Subnet)

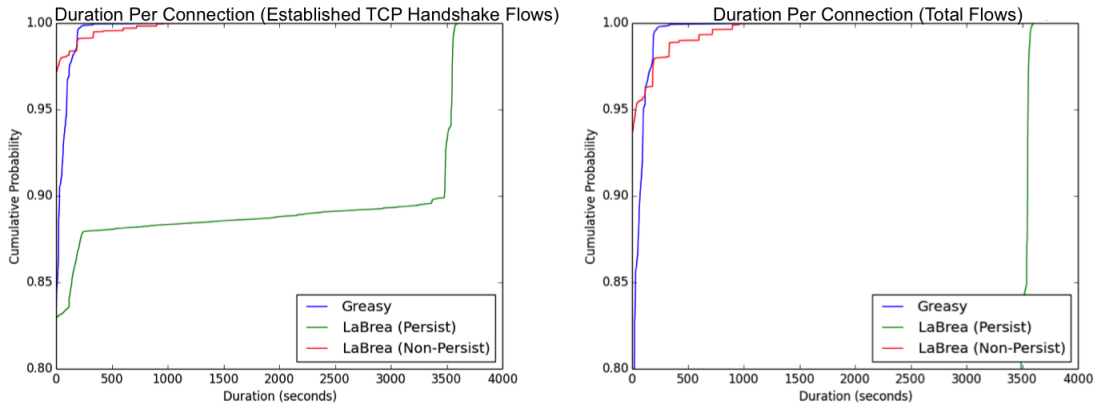

Figure 4.16: CDF of Duration Per Connection (Same /24 Subnet)

We observe that the results for the number of packets both received and sent by the tarpit applications are the same for each tarpit. *LaBrea* in persist mode sent and received on average 20 packets per flow, *Greasy* 7 packets packets per flow, and *LaBrea* in non-persist mode 2 packets per flow. Figure 4.17, Figure 4.18, Figure 4.19, and Figure 4.20 show the distribution of packets for each of the three tarpit applications/modes. These results match our earlier observations for number of packets per flow.

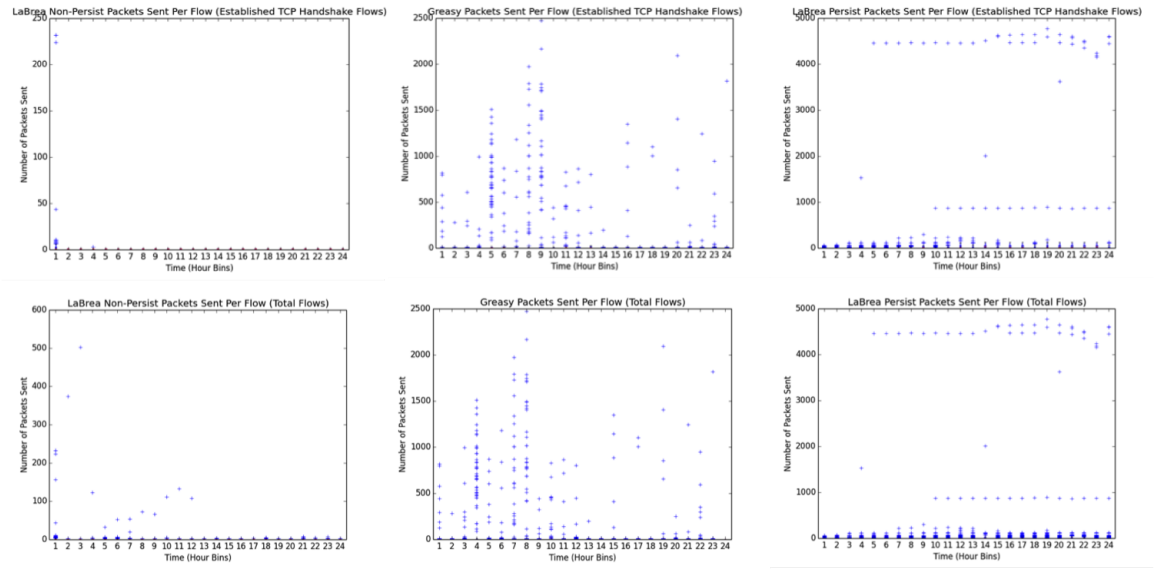

Figure 4.17: Boxplot of Packets Sent per Flow (Same /24 Subnet)

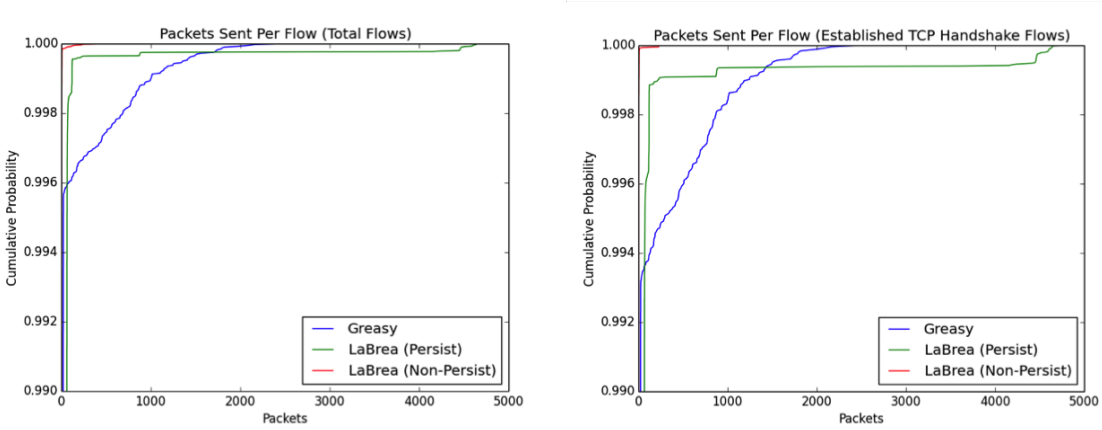

Figure 4.18: CDF of Packets Sent per Flow (Same /24 Subnet)

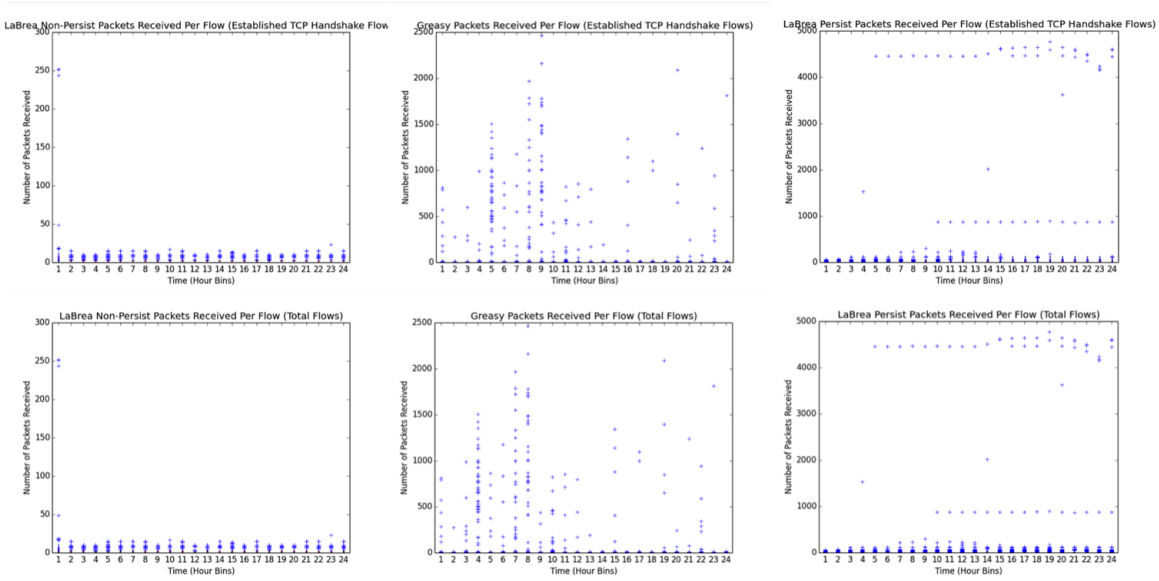

Figure 4.19: Boxplot of Packets Received per Flow (Same /24 Subnet)

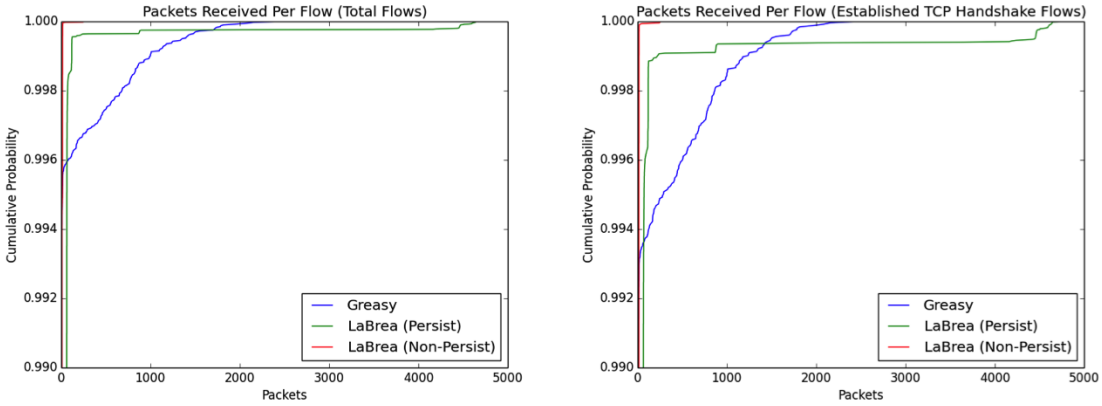

Figure 4.20: CDF of Packets Received per Flow (Same /24 Subnet)

The results from our experiments measuring each tarpit's degree of stickiness indicate that there is a correlation between the duration of a connection and the number of packets transmitted during a connection. Because the duration of a connection is quite indicative of the tarpit's stickiness, we can infer that the number of packets transmitted during the connection also affects the tarpit's stickiness. *LaBrea* in persist mode sends and receives the most packets and holds the longest connections on average, followed by *Greasy*. *LaBrea*

in non-persist mode comes in third in terms of stickiness.

4.3.3 Packet Latency

We ran the packet latency experiment three times and collected three sets of RTT measurements from each vantage point-subnet pair. For each pair, we calculated the average RTT and standard deviation. The RTT values are shown in Tables 4.2, 4.3, and 4.4. Then for each vantage point, we calculated the percent of change of the tarpit sample mean compared to the control mean. Using the z-statistic model described in Section 4.1.3 we calculated the z-statistic values as shown in Tables 4.5, 4.6, and 4.7. Overall both *LaBrea* and *Greasy* had z-values much smaller than our left-tail critical value -2.58 and much greater than our right-tail critical value 2.58, respectively. *LaBrea* had an average z-value of -39.40 and *Greasy* an average z-value of 25.07. We were able to reject our null hypothesis with 99% confidence indicating that both tarpit samples represented different populations from the control. The one exception is the z-statistic from the third run using our residential vantage point, where *LaBrea* had a z-value of 0.05 between -2.58 and 2.58, well within our null hypothesis, shown in Table 4.7. But because this value was only one out of the nine calculated z-values for *LaBrea* that was within range of our critical values, we went with the majority of *LaBrea*'s z-values. As expected, the closer the vantage point is to the target subnet, the smaller the RTT values. It is interesting to note that the largest and percent of change values for *Greasy* and *LaBrea*, respectively, are from observed from our closest vantage point, the local subnet. This is further observed in the Probability Distribution Function (PDF) of RTT measurements from each of the runs using each vantage point. Figure 4.21, Figure 4.22, and Figure 4.23 show the PDFs of results using our residential vantage point. The distributions are fairly similar in these Figures, with *LaBrea*'s distribution shifted slightly to the left of the control distribution, and *Greasy*'s distribution shifted slightly to the right; RTT values are generally between 100-200 ms. Figure 4.22 shows *Greasy* with a wider range of RTT values, resulting in a wider curve. Figure 4.24, Figure 4.25, and Figure 4.26 show PDFs of results from our NPS vantage point. Again, we see *LaBrea*'s distribution located more to the left of the control curve, and *Greasy*'s distribution located only slightly more to the right of the control distribution. RTT values range from 80-90ms. The PDFs of results from our local subnet vantage point show the greatest variability in distribution, as shown in Figure 4.27, Figure 4.28, and Figure 4.29. The range of RTT values for both

Greasy and *LaBrea* lie within the range of values of the control, but the distributions are distinct from that of the control. Although the z-values reveal that both *LaBrea* and *Greasy* represent different populations from our control at a millisecond-level of granularity, Tables 4.2, 4.3, and 4.4 show that both *LaBrea* and *Greasy*'s RTT values differ only by less than a second, which is not a noticeable difference.

Table 4.2: Average RTT Measurements from Run 1

Vantage Points	Control	*LaBrea*	*Greasy*
Residential	114.24 ms.	116.28 ms.	116.97 ms.
NPS	82.23 ms.	81.48 ms.	82.56 ms.
Local	1.26 ms.	0.51 ms.	1.65 ms.

Table 4.3: Average RTT Measurements from Run 2

Vantage Points	Control	*LaBrea*	*Greasy*
Residential	135.87 ms.	128.92 ms.	133.77 ms.
NPS	82.37 ms.	81.56 ms.	82.64 ms.
Local	1.41 ms.	0.52 ms.	1.63 ms.

Table 4.4: Average RTT Measurements from Run 3

Vantage Points	Control	*LaBrea*	*Greasy*
Residential	127.24 ms.	127.25 ms.	130.54 ms.
NPS	82.30 ms.	81.53 ms.	82.95 ms.
Local	1.25 ms.	0.51 ms.	1.66 ms.

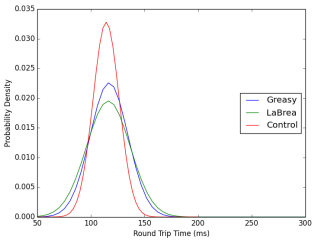

Figure 4.21: PDF of RTT Measurements from Run 1 of Latency Experiment Using Residential Vantage Point.

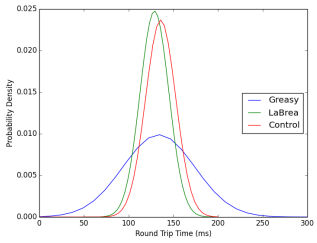

Figure 4.22: PDF of RTT Measurements from Run 2 of Latency Experiment Using Residential Vantage Point.

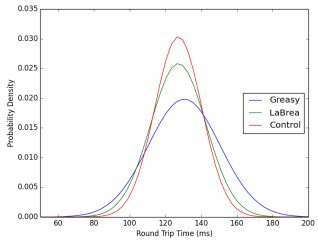

Figure 4.23: PDF of RTT Measurements from Run 3 of Latency Experiment Using Residential Vantage Point.

Table 4.5: Z-statistic and Percent of Change Results from Run 1 of Latency Experiment

Vantage Points	*LaBrea* % Change	*LaBrea* SE	*Greasy* % Change	*Greasy* SE
Residential	1.78%	16.40	2.39%	21.97
NPS	-0.92%	-248.11	0.40%	108.92
Local	-59.32%	28.98	31.05%	15.16

Table 4.6: Z-statistic and Percent of Change Results from Run 2 of Latency Experiment

Vantage Points	*LaBrea* % Change	*LaBrea* SE	*Greasy* % Change	*Greasy* SE
Residential	-5.12%	-20.62	-1.55%	-6.16
NPS	-0.98%	-35.42	0.33%	11.85
Local	-63.22%	-24.39	15.66%	6.04

Table 4.7: Z-statistic and Percent of Change Results from Run 3 of Latency Experiment

Vantage Points	*LaBrea* % Change	*LaBrea* SE	*Greasy* % Change	*Greasy* SE
Residential	0.01%	0.05	2.60%	15.48
NPS	-0.93%	-43.74	0.78%	36.79
Local	-59.05%	-27.76	33.08%	15.55

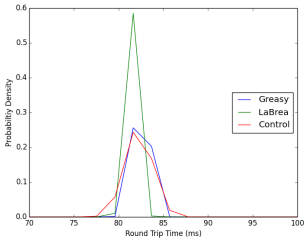

Figure 4.24: PDF of RTT Measurements from Run 1 of Latency Experiment Using NPS Vantage Point.

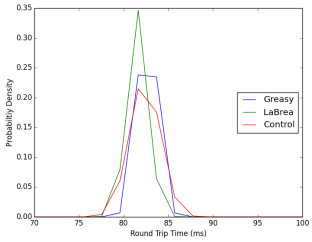

Figure 4.25: PDF of RTT Measurements from Run 2 of Latency Experiment Using NPS Vantage Point.

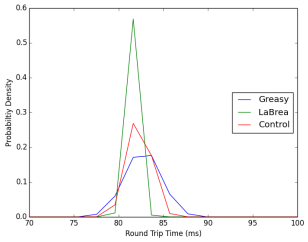

Figure 4.26: PDF of RTT Measurements from Run 3 of Latency Experiment Using NPS Vantage Point.

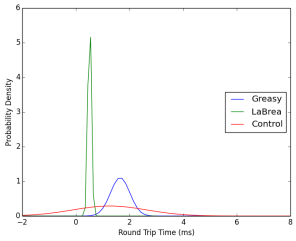

Figure 4.27: PDF of RTT Measurements from Run 1 of Latency Experiment Using Local Subnet Vantage Point.

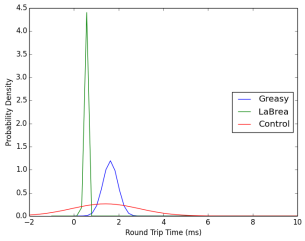

Figure 4.28: PDF of RTT Measurements from Run 2 of Latency Experiment Using Local Subnet Vantage Point.

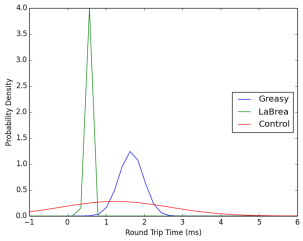

Figure 4.29: PDF of RTT Measurements from Run 3 of Latency Experiment Using Local Subnet Vantage Point.

CHAPTER 5:
Conclusion and Future Work

5.1 Conclusion

Using the experiments discussed above, we measured *Greasy*'s ability to deceive *Degreaser*, degree of stickiness compared to *LaBrea*, and potential processing overhead as observed by packet latency. *Degreaser* was only able to successfully identify 2.8% of *Greasy*'s tarpit hosts as tarpits. This indicates that we effectively mitigated the two main tarpit signatures used by *Degreaser*'s tarpit identification heuristics: lack of TCP options in TCP packets sent by the tarpit, and TCP receive window size.

One of our biggest contributions for this thesis was to run *Greasy* in production on an Internet-facing /24 subnet. We were able to receive real traffic on the network and study *Greasy*'s ability to handle and interact with a bombardment of various scans and probes from unknown remote sources. Our stickiness experiments revealed that *Greasy* was able to tarpit a connection longer than *LaBrea* in non-persist mode. *Greasy* sent 6 times more packets than *LaBrea*, and kept the connection alive 7 times longer than *LaBrea*. *LaBrea* in persist mode outperformed *Greasy* in stickiness. *LaBrea* sent around 6 times more packets than *Greasy*, and held the connection for a significantly longer duration than *Greasy*– around 50 times longer. *LaBrea* in persist mode is able to keep a stickier connection because, according to the TCP protocol, it can hold a persistent connection for an indefinite amount of time until the remote end disconnects, or deadlock indefinitely [34] as discussed in Chapter 2. For instance, range of values for duration distribution for *LaBrea* in persist mode span from 0 to 3600 seconds, which equals 1 hour. Because we bin our results by hours, it is possible there are connections held indefinitely that are missed in our analysis because the TCP handshake was established in an earlier bin. Although *LaBrea* in persist mode is able to keep a connection tarpitted longer than *Greasy*, it is easily detected because of its predictable constant zero window size. *Greasy*, on the other hand, incorporates persist mode, along with three other tarpitting modes to create random tarpitting behavior. So although *Greasy* may not hold the stickiest connection, it successfully improves

its tarpitting capabilities, while still evading detection. A simple solution to increasing the stickiness of *Greasy*'s connections is to alter its tarpitting algorithm to increase the probability that persist mode is chosen, but at a rate that still ensures *Greasy* will not be detected. Our experiment to measure the degree of stickiness of a tarpit showed that there is a loose correlation between the number of packets transferred within a tarpit connection and the total duration of the connection. The longer the duration of a connection, the more packets transferred in that connection. However, this is only a loose correlation because it is possible that connections are held open for a long duration while only a few packets are transferred between two endpoints.

We used the packet latency tests to measure the amount of overhead, as observed by packet latency, that *Greasy* may incur due to its implementations and inclusion of the *libtins* sniffer. This experiment revealed that both *Greasy* and *LaBrea* have ICMP RTT mean values that lie outside the normal range of control RTT means at the millisecond level of granularity. We used a two-tailed hypothesis test to test for extreme values from the mean, and chose critical values -2.58 and 2.58 to decide whether to accept or reject the null hypothesis that the control and tarpit samples are the same population (i.e., normal hosts and tarpit hosts are indistinguishable by packet latency). *Greasy* had z-values much larger than 2.58, an average z-value of 25.07, and *LaBrea* had z-values much smaller than -2.58, an average z-value of -39.40. Thus, we reject the null hypothesis with a 99% confidence level. However, despite rejecting the null hypothesis, the actual mean RTT values for *Greasy* and *LaBrea* differ from the control mean RTTs by about 1 ms, which is not a noticeable difference, so packet latency as measured using ICMP RTTs is negligible.

In addition to developing a tarpit with overall improved tarpit performance, we also provide an extensible platform on which to further develop and advance *Greasy*'s tarpitting capabilities.

5.2 Future Work

5.2.1 Improvements on Current Implementations

Greasy's current implementation maintains state to operate the duplicate ACK tarpitting behavior. This could lead to performance degradation and scalability issues if the machine

running *Greasy* exhausts its memory due to the large volume of connections *Greasy* must remember. Ideally a modified implementation will make use of TCP/IP header values and the SYN cookie-like hashing algorithm to maintain minimal state.

Another improvement is to increase the stickiness of *Greasy*'s tarpit. Currently, its degree of stickiness lies between that of *LaBrea* in non-persist mode and *LaBrea* in persist mode. We try to balance the line between detectability and deceptiveness versus improving tarpitting capabilities. Our experiments show that *LaBrea* in persist mode holds the stickiest connections but is also trivial to detect because of its constant and predictable zero window. One of our main goals is to build application-level tarpitting capabilities in *Greasy*, so perhaps the level of stickiness can improve at the application-level vice TCP level to make the tarpit both less detectable and stickier.

5.2.2 Additional Features

Greasy provides a foundational and extensible platform for future tarpit development. The three main areas we would like to extend *Greasy* is in IPv6 compatibility, operating system portability, and Application-Layer tarpitting functionality to create tarpit profiles that correlate open ports, TCP options and MAC addresses into a believable Operating System (OS) stack.

IPv6

Adversaries target both IPv4 and IPv6 hosts in their scanning operations, so we want to extend *Greasy*'s functionality to also handle IPv6 packets. The 128-bit IPv6 address increased the difficulty for a scanner to locate vulnerable targets using IPv4 random scanning techniques [74]. However, worms may be able to overcome this large search space and continue their scanning operations in IPv6 space by leveraging weaknesses in certain IPv6 features. For instance, unlike IPv4 which uses ARP to resolve IP addresses to MAC addresses on a LAN, IPv6 uses a Neighbor Discovery [75] protocol. Through this mechanism, adversaries may be able to consult host routing tables or passively listen as a participant in the routing protocol to identify live hosts on the LAN. IPv6 also uses multicast to send packets to all hosts on the LAN. Adversaries may abuse this feature by sending multicast messages to find all routers or hosts on the LAN [74].

We must add a new module in *Greasy* to handle IPv6 soft-capturing, since IPv6 uses the Neighbor Discovery protocol instead of ARP for IP to MAC addresses mapping. The Neighbor Discovery protocol uses Neighbor Solicitation messages to resolve link-layer addresses of neighboring nodes or confirm neighbor reachability [75]. Neighbor Advertisement messages are used in response to Solicitation messages, and may also be used to announce link-layer address changes, similar to the gratuitous ARP. While waiting for Advertisement messages, the requesting node should retransmit Neighbor Solicitation messages every *RetransTimer* milliseconds. If no Neighbor Advertisement message is received after *MAX_MULTICAST_SOLICIT* solicitations, then address resolution failed, and an Internet Control Message Protocol Version 6 (ICMPv6) host unreachable message is returned. *Libtins* supports both implementations of IPv6 and ICMPv6 packet crafting and handling, so *Greasy* will be able to easily add soft-capturing of IPv6 packets using this Neighbor Discovery protocol [75]. A new module for TCP packet handling of IPv6 hosts must also be added to *Greasy* since deterministic random hashing functions have to be modified to cater to the IPv6 address structure.

Portability

We want to be able to deploy *Greasy* on any given operating system to cater to users' preferences and allow for wider deployment of this tarpit. *Greasy*'s current implementation is compatible with *FreeBSD* and *Ubuntu Linux distribution* platforms. It has not been tested on other operating systems. The *Libtins* packet crafting library is portable on *Windows*, *OSX*, in addition to little and big endian *GNU/Linux* and *FreeBSD* operating systems [61]. *Libtins* has special instructions for its packet sniffing and sending functions when running applications on *Windows*, but for the most part, the underlying code structure of *Greasy* should not change from operating system to operating system.

Application Layer Tarpitting

Besides TCP-level tarpitting, we can leverage specific features of certain protocols to create an application specific tarpit aimed at producing a more realistic and stickier tarpit product. We hope to extend *Greasy*'s tarpitting capabilities by adding application layer tarpitting in addition to its current TCP layer tarpitting. Motivation for expanding *Greasy*'s functionality to handle application-level tarpitting stems from a couple of reasons. One reason is to

include application-level tarpitting for realism. We want to generate responses an adversary expects to see when connecting to a particular service to help the adversary believe the tarpit is a real host. We also want to have actual data to send in order to maintain a more persistent connection without being detected. In addition, we want to allow better integration with existing honeypots that currently perform application-layer functionalities. Several suggested application layer protocols that *Greasy* should tarpit include Secure Shell (SSH), SMTP, and HTTP, among others.

SSH Tarpitting : SSH [76] is an encrypted protocol used to create secure remote log-in sessions and other network services such as forwarded TCP/IP and X11 connections. SSH servers that are not public facing require the remote client to enter login credentials (i.e., username and password) in order to authenticate to the server. One method of tarpitting the remote client is to add a configurable delay between the client's initial SSH login attempt and the sending of the login prompt.

SMTP Tarpitting : Similar to SMTP tarpits [43] introduced in Chapter 2, we can leverage certain features like continuation lines in order to tarpit adversaries attempting to send bulk spam. The goal is to make use of SMTP's continuation line scheme to clog up the connection by sending tons of continuation lines with fake messages. This forces the remote end to continue to wait until the multi-line message is finished. As explained in Chapter 2, this method alone is not at all effective against an adversary as they are able to parallelize the process, and existing SMTP tarpits can only tarpit one connection at a time. We can take advantage of *Greasy*'s multi-threading capabilities to try to parallelize this tarpitting capability, as well as implement our hashing algorithms to keep track of tarpitted SMTP connections without maintaining state.

HTTP Tarpitting : Adversaries dig through web pages for information such as email addresses and domain names to look for vulnerable hosts [43] in addition to IP scanning. Similar to HTTP tarpits discussed in Chapter 2, we can foil an adversary in their attempt to gather information via webpages by creating random webpages with fake information and allow *Greasy* to tarpit these HTTP connections.

Border Gateway Protocol (BGP) Tarpitting : BGP [77] is used to exchange routing and reachability information among Autonomous Systems (ASs) [78] (i.e., groups of connected

IP prefixes). BGP routing worms take advantage of BGP routing tables to learn about the distribution of vulnerable hosts in a particular routing space. A suggested BGP tarpit implementation includes the addition of fake entries in the BGP routing tables to create the illusion of the existence of vulnerable hosts. This will cause the adversary to spend additional time interacting with these fake hosts.

Domain Name System (DNS) Tarpitting : DNS [79] is used to resolve domain names to numerical IP addresses, and vice versa. Adversaries take advantage of the DNS infrastructure to guess DNS names instead of IP addresses in order to find likely vulnerable targets [80]. Additionally, despite the greater costs (e.g., time) of issuing a DNS query compared to randomly scanning IP addresses, Kamra *et al.* found that DNS random IPv6 scanning produced worm propagation speeds comparable to that of traditional random IPv4 scanning [80]. We can easily deceive adversaries by associating fake domain names to *Greasy*'s pool of fake IP addresses. Althogh this tarpit may be useful, it is currently out of the scope of *Greasy*'s current implementation because it would require implementation of UDP tarpitting.

List of References

[1] V. Yegneswaran, P. Barford, and J. Ullrich, "Internet intrusions: global characteristics and prevalence," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 138–147, 2003.

[2] A. Dainotti, A. King, and K. Claffy, "Analysis of Internet-wide probing using darknets," in *Proceedings of the 2012 ACM Workshop on Building analysis datasets and gathering experience returns for security*. ACM, 2012, pp. 13–14.

[3] M. A. Rajab, F. Monrose, and A. Terzis, "On the Effectiveness of Distributed Worm Monitoring." in *Usenix Security*, 2005.

[4] T. Olzak, "The five phases of a successful network penetration," TechRepublic, 2008. [Online]. Available: http://www.techrepublic.com/blog/it-security/the-five-phases-of-a-successful-network-penetration/

[5] L. Alt, R. Beverly, and A. Dainotti, "Uncovering network tarpits with Degreaser," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 156–165.

[6] T. Liston, "LaBrea," 2003. [Online]. Available: http://labrea.sourceforge.net/labrea.1.txt

[7] Y. Song, S. Shin, and Y. Choi, "Network Iron Curtain: Hide Enterprise Networks with Openflow," in *Information Security Applications*. Springer, 2014, pp. 218–230.

[8] D. Moore, C. Shannon *et al.*, "Code-Red: a case study on the spread and victims of an Internet worm," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 2002, pp. 273–284.

[9] S. Shin and G. Gu, "Conficker and beyond: a large-scale empirical study," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 151–160.

[10] Z. Durumeric, M. Bailey, and J. A. Halderman, "An Internet-wide view of Internet-wide scanning," in *USENIX Security Symposium*, 2014.

[11] V. Yegneswaran, P. Barford, and D. Plonka, "On the design and use of Internet sinks for network abuse monitoring," in *Recent Advances in Intrusion Detection*. Springer, 2004, pp. 146–165.

[12] N. C. Rowe and H. C. Goh, “Thwarting cyber-attack reconnaissance with inconsistency and deception,” in *Information Assurance and Security Workshop, 2007. IAW’07. IEEE SMC*. IEEE, 2007, pp. 151–158.

[13] F. Cohen, “A note on the role of deception in information protection,” *Computers & Security*, vol. 17, no. 6, pp. 483–506, 1998.

[14] D. Evans, A. Nguyen-Tuong, and J. Knight, “Effectiveness of moving target defenses,” in *Moving Target Defense*. Springer, 2011, pp. 29–48.

[15] A. Barfar and S. Mohammadi, “Honeypots: intrusion deception,” *ISSA Journal*, pp. 28–31, 2007.

[16] L. Spitzner, “The Honeynet Project: Trapping the hackers,” *IEEE Security & Privacy*, no. 2, pp. 15–23, 2003.

[17] S. O. Hunter, “Virtual Honeypots: Management, attack analysis and democracy,” 2010.

[18] C. Ruvalcaba, “Smart IDS – Hybrid LaBrea Tarpit,” SANS Institute, Report, 2009.

[19] V. Oppleman, “Network Defense Applications using IP Sinkholes.”

[20] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling, “The nepenthes platform: An efficient approach to collect malware,” in *Recent Advances in Intrusion Detection*. Springer, 2006, pp. 165–184.

[21] H. Artail, H. Safa, M. Sraj, I. Kuwatly, and Z. Al-Masri, “A hybrid honeypot framework for improving intrusion detection systems in protecting organizational networks,” *computers & security*, vol. 25, no. 4, pp. 274–288, 2006.

[22] R. Joshi and A. Sardana, *Honeypots: A New Paradigm to Information Security*. CRC Press, 2011.

[23] T. Liston, “LaBrea: "sticky" Honeypot and IDS.” [Online]. Available: http://labrea.sourceforge.net/labrea-info.html

[24] F. Pouget and M. Dacier, “White paper: Honeypot, honeynet: A comparative survey,” Technical Report RR-03-082, Institut Eurecom, Tech. Rep., 2003.

[25] J. Kristoff, “TCP Congestion Control,” *Tech Notes*, 2002.

[26] T. Socolofsky and C. Kale, “A TCP/IP Tutorial,” RFC 1180, Internet Engineering Task Force, 1991. [Online]. Available: https://tools.ietf.org/html/rfc1180

[27] L. Haig, “LaBrea–A New Approach to Securing our Networks,” SANS Institute, Report, 2002.

[28] R. Braden, “Requirements for Internet Hosts – Communication Layers,” RFC 1122, Internet Engineering Task Force, 1989. [Online]. Available: https://tools.ietf.org/html/rfc1122

[29] S. Cheshire, “IPv4 Address Conflict Detection,” RFC 5227, Internet Engineering Task Force, 2008. [Online]. Available: https://tools.ietf.org/html/rfc5227

[30] H. Arora, “Hubs vs Switches vs Routers – Networking Device Fundamentals,” The Geek Stuff, 2013.

[31] “Transmission Control Protocol,” RFC 793, Internet Engineering Task Force, 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[32] J. Chu, N. Dukkipati, Y. Cheng, and M. Mathis, “Increasing TCP’s Initial Window,” RFC 6928, Internet Engineering Task Force, 2013. [Online]. Available: https://tools.ietf.org/html/rfc6928

[33] M. Bashyam, M. Jethanandani, and A. Ramaiah, “TCP Sender Clarification for Persist Condition,” RFC 6429, Internet Engineering Task Force, 2011. [Online]. Available: https://tools.ietf.org/html/rfc6429

[34] K. R. Fall and W. R. Stevens, *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011.

[35] J. Freniche, “TCP Window Probe Deadlock,” Internet Engineering Task Force, 1998. [Online]. Available: https://tools.ietf.org/html/draft-rfced-info-freniche-00

[36] M. Allman, V. Paxson, and W. R. Stevens, “TCP Congestion Control,” RFC 2581, Internet Engineering Task Force, Apr. 1999. [Online]. Available: https://tools.ietf.org/html/rfc2581#section-3.2

[37] S. Floyd, T. Hnderson, and A. Gurtov, “The NewReno Modification to TCP’s Fast Recovery Algorithm,” RFC 3782, Internet Engineering Task Force, Apr. 2004. [Online]. Available: https://www.ietf.org/rfc/rfc3782.txt

[38] T. Holz and F. Raynal, “Detecting honeypots and other suspicious environments,” in *Information Assurance Workshop, 2005. IAW’05. Proceedings from the Sixth Annual IEEE SMC*. IEEE, 2005, pp. 29–36.

[39] A. Hopkins, “TARPIT-iptables TARPIT target.” [Online]. Available: http://www.netfilter.org/projects/patch-o-matic/pom-external.html

[40] S. Mukkamala, K. Yendrapalli, R. Basnet, M. Shankarapani, and A. Sung, "Detection of virtual environments and low interaction honeypots," in *Information Assurance and Security Workshop, 2007. IAW'07. IEEE SMC*. IEEE, 2007, pp. 92–98.

[41] K. Gubbels, "Hands in the honeypot," *GIAC Security Essentials Certification (GSEC)*, 2002.

[42] T. Eggendorfer, "Reducing spam to 20% of its original value with a SMTP tar pit simulator," in *In MIT Spam Conference*. Citeseer, 2007.

[43] T. Eggendorfer, "A proposal for an efficient way to prevent spam by analysing SMTP and HTTP tar pits towards their efficiency in fighting spam and combining them," in *Proceedings of the 10th WSEAS international conference on Communications*. World Scientific and Engineering Academy and Society (WSEAS), 2006, pp. 449–454.

[44] J. B. Postel, "Simple Mail Transfer Protocol," RFC 821, Internet Engineering Task Force, 1982. [Online]. Available: https://tools.ietf.org/html/rfc821

[45] D. J. Bernstein, "SYN cookies," 1996.

[46] "TCP Option Kind Numbers," IANA. [Online]. Available: http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#tcp-parameters-1

[47] [Online]. Available: http://pastebin.com/u5YWvbVw

[48] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee, "Using prime numbers for cache indexing to eliminate conflict misses," in *Software, IEE Proceedings-*. IEEE, 2004, pp. 288–299.

[49] M. Zalewski, "p0f - fingerprint database." [Online]. Available: https://github.com/p0f/p0f/blob/master/p0f.fp

[50] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgement Options," RFC 2018, Internet Engineering Task Force, 1996. [Online]. Available: https://tools.ietf.org/html/rfc2018

[51] D. Eastlake 3rd and J. Abley, "IANA Considerations and IETF Protocol and Documentation Usage for IEEE 802 Parameters," RFC 7042, Internet Engineering Task Force, 2013. [Online]. Available: https://tools.ietf.org/html/rfc7042

[52] "Guidelines for 48-bit Global Identifier (EUI-48)," IEEE Standards Association. [Online]. Available: https://standards.ieee.org/develop/regauth/tut/eui48.pdf

[53] “IEEE Standards Association Registration Authority.” [Online]. Available: https://regauth.standards.ieee.org/standards-ra-web/pub/view.html#registries

[54] R. Rivest, “The MD5 Message-Digest Algorithm,” RFC 1321, Internet Engineering Task Force, 1992. [Online]. Available: https://www.ietf.org/rfc/rfc1321.txt

[55] EC-Council, *Penetration Testing: Network Threat Testing*. USA: Cengage Learning, 2010.

[56] “Appendix C. Common Ports.” [Online]. Available: http://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-sg-en-4/ch-ports.html

[57] G. Huston, “IPv4: How long do we have?” *The Internet Protocol Journal*, vol. 6, no. 4, pp. 2008–2010, 2003.

[58] “Internet-Wide Scan Data Repository,” 2014. [Online]. Available: https://scans.io/study/sonar.http

[59] H.-J. Shiau, “Understanding Gratuitous ARPs,” 2014.

[60] “Internet Control Message Protocol,” RFC 792, Internet Engineering Task Force, 1981. [Online]. Available: https://tools.ietf.org/html/rfc792

[61] M. Fontanini, “Libtins: packet crafting and sniffing library.” [Online]. Available: http://libtins.github.io

[62] “Libconfig - C/C++ Configuration File Library.” [Online]. Available: http://www.hyperrealm.com/libconfig/

[63] “Easylogging.” [Online]. Available: https://github.com/easylogging/easyloggingpp

[64] B. Barney, “POSIX Threads Programming,” Lawrence Livermore National Laboratory, 2015.

[65] B. Claise, “Cisco Systems NetFlow Services Export Version 9,” RFC 3954, Internet Engineering Task Force, 2004. [Online]. Available: http://tools.ietf.org/html/rfc3954.html

[66] T. Nadeau, M. Morrow, G. Swallow, D. Allan, and S. Matasushima, “Operations and Management (OAM) Requirements for Multi-Protocol Label Switched (MPLS) Networks,” RFC 4377, Internet Engineering Task Force, 2006. [Online]. Available: https://tools.ietf.org/html/rfc4377

[67] Z. Trabelsi, H. Rahmani, K. Kaouech, and M. Frikha, "Malicious sniffing systems detection platform," in *Applications and the Internet, 2004. Proceedings. 2004 International Symposium on*. IEEE, 2004, pp. 201–207.

[68] M. A. Sportack, *TCP/IP first-step*. Pearson Education, 2004.

[69] "Z-score: Definition, Formula and Calculation." [Online]. Available: http://www.statisticshowto.com/how-to-calculate-a-z-score/

[70] "promiscuous mode." [Online]. Available: http://searchsecurity.techtarget.com/definition/promiscuous-mode

[71] B. Irwin, "Network Telescope Metrics," in *Southern African Telecommunications and Applications Conference (SATNAC)*, 2012.

[72] "TCPDUMP & LIBPCAP." [Online]. Available: http://www.tcpdump.org

[73] M. Allman, V. Paxson, and J. Terrell, "A brief history of scanning," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 77–82.

[74] S. M. Bellovin, B. Cheswick, and A. Keromytis, "Worm propagation strategies in an IPv6 Internet," *LOGIN: The USENIX Magazine*, vol. 31, no. 1, pp. 70–76, 2006.

[75] T. Narten, E. Nordmark, W. Simpson, and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)," RFC 4861, Internet Engineering Task Force, 2007. [Online]. Available: https://tools.ietf.org/html/rfc4861

[76] T. Ylonen and C. Longvick, "The Secure Shell (SSH) Transport Layer Protocol," RFC 4253, Internet Engineering Task Force, 2006. [Online]. Available: https://tools.ietf.org/html/rfc4253

[77] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," RFC 4271, Internet Engineering Task Force, 2006. [Online]. Available: https://tools.ietf.org/html/rfc4271

[78] J. Hawkinson and T. Bates, "Guidelines for creation, selection, and registration of an Autonomous System (AS)," RFC 1930, Internet Engineering Task Force, 1996. [Online]. Available: https://tools.ietf.org/html/rfc1930

[79] P. Mockapetris, "Domain Names - Concepts and Facilities," RFC 1034, Internet Engineering Task Force, 1987. [Online]. Available: https://tools.ietf.org/html/rfc1034

[80] A. Kamra, H. Feng, V. Misra, and A. D. Keromytis, "The effect of DNS delays on worm propagation in an IPv6 Internet," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 4. IEEE, 2005, pp. 2405–2414.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California