Towards Understanding Application Semantics of Network Traffic

Ruoming Pang

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

Advisor: Larry L. Peterson

June 2008

Abstract

This dissertation explores the problem of building a *semantic network traffic analysis* system and using it to investigate various aspects of network traffic. *Semantic* traffic analysis uncovers the application-layer semantics conveyed in packets so that one can examine the specific requests, responses, status messages, error codes, and data items embedded in a connection dialog. Analyzing these at the application layer, as opposed to the syntactic byte-string layer, opens up much greater insight into the nature and context of the exchange between two hosts. For this reason, semantic traffic analysis is a cornerstone for precise network intrusion detection and also has broad applications in measurements of networking systems.

This dissertation advances semantic traffic analysis in two aspects. First, we present tools and techniques for building traffic analyzers and creating shared traces, including design and implementation of (1) a declarative language, `binpac`, for writing application protocol parsers, and (2) a programming environment for packet trace transformation and anonymization. Second, we characterize two types of previously unstudied network traffic, Internet background radiation and enterprise internal traffic. Both studies focus on traffic semantics, aiming to understand the network applications that generate the traffic and to uncover underlying causes of network usage patterns.

Acknowledgments

I enjoyed my graduate study at Princeton so much that I hardly wanted to graduate. I would like to thank many people for the great time.

First I am grateful to have Prof. Larry Peterson as my advisor. Besides being an exemplary advisor with great vision and gentle guidance, Larry also gave me a lot of freedom in my research and supported me through the years.

I am very fortunate to have the opportunity to work closely with Dr. Vern Paxson, first as a summer intern at ICIR, then on a number of fascinating projects that eventually lead to this dissertation. I am most impressed by Vern's abundance of research ideas and rigorous treatment of measurements.

I want to thank my other dissertation committee members, not only for serving on the committee, but also for going out of their way to help me finish this dissertation: Prof. Jennifer Rexford, for her patience in reading through my lengthy thesis multiple times and her insightful and constructive comments, Prof. Brian Kernighan, for nagging me to finish my thesis whenever we meet at Google and for volunteering to proof-read my thesis, and Prof. Vivek Pai, for teaching me how to build robust distributed systems when we worked on CoDeeN.

I also want to thank other Princeton faculty and staff. I am especially grateful to Prof. Randy Wang for attracting me to the field of computer systems, Prof. Sanjeev Arora and Prof. Amit Sahai for introducing me to the captivating world of theoretic computer science, and last but not least, Melissa Lawson, for bringing efficiency and warmth to the department.

I also would like express my deep appreciation to my collaborators and colleagues, from whom I learned many things: Robin Sommer, Vinod Yegneswaran, Paul Barford, Mark Allman, Christian Kreibich, Yuqun Chen, Lujo Bauer, David Walker, Ed Felten, Limin Jia, Xiaohu Qie, Liming Wang, KyoungSoo Park, Aki Nakao, Scott Karlin, Tammo Spalink, Andy Bavier, Mark Huang, Marc Fiuczynski, and staff at Lawrence Berkeley National Laboratory.

I dedicate this dissertation to my parents, my grandmother, and my wife, Zhiyan Liu, for making me who I am.

This dissertation was supported by grants from NSF Awards CNS-0520053, CCR-0325653, ITR/ANI-0205519, EIN-0335214, 0335241, 0205519, 0433702, ANI-9906704, STI-0334088, ARO grant DAAD19-02-1-0304, DHS grant HSHQPA4X03322, DARPA contract F30602–00–2–0561, and the Intel Corporation.

Contents

Abstract . . . iii

List of Figures . . . ix

List of Tables . . . xi

1 Introduction **1**

1.1 Background and Motivation . . . 1

1.2 Challenges . . . 3

1.3 Dissertation Contributions . . . 4

1.3.1 Tools and Techniques . . . 4

1.3.2 Measurement Studies . . . 6

1.3.3 Thesis Outline . . . 8

2 Traffic Semantic Analysis and `bro` **9**

2.1 Traffic Capture . . . 9

2.2 Extracting High Level Traffic Representation . . . 9

2.2.1 TCP/IP Processing . . . 10

2.2.2 Recognizing Applications . . . 10

2.2.3 Parsing Application Protocols . . . 11

2.3 Traffic Analysis and Manipulation . . . 12

2.4 Summary . . . 15

3 `binpac`—A `yacc` for Writing Application Protocol Parsers **16**

3.1 Problem Statement . . . 17

3.2 Related Work 19

3.3 Characteristics of Application Protocols 21

3.3.1 Syntax and Grammar Issues 21

3.3.2 Concurrent Input 23

3.3.3 Robustness 23

3.4 The `binpac` Language 24

3.4.1 Data Model 24

3.4.2 State Management 33

3.4.3 Integrating Custom Computation 36

3.4.4 Separation of Concerns 36

3.5 Parser Generation 37

3.5.1 Incremental Input 37

3.5.2 Error Detection and Recovery 38

3.6 Experiences with `binpac` 41

3.7 Summary and Future Directions 42

4 A Programming Environment for Trace Anonymization **44**

4.1 Problem Statement 45

4.2 Goals 47

4.3 Generic Trace Transformation 48

4.3.1 Policy Script Programming Environment 48

4.3.2 Trace Parsing 50

4.3.3 Trace Composer 51

4.3.4 Trace Rewriters for Trace Size Reduction 56

4.4 Trace Anonymization 57

4.4.1 Objectives of Anonymization 57

4.4.2 Anonymization Primitives 57

4.4.3 Inference Attacks 59

4.5 Case Study: FTP Anonymization 61

4.5.1 The Filter-In Principle 62

4.5.2 Selected Details of FTP Anonymization 63
4.5.3 Refining with Manual Inspection 64
4.5.4 Reply Anonymization 66
4.5.5 Verification 68
4.5.6 Discussion 69
4.6 Related Work 71
4.7 Summary 72

5 Characteristics of Internet Background Radiation 73

5.1 Problem Statement 74
5.2 Related Work 75
5.3 Measurement Methodology 76
5.3.1 Application-Level Responders 77
5.3.2 Taming the Traffic Volume 81
5.3.3 Traffic Analysis 84
5.3.4 Experimental Setup 84
5.4 Passive Measurement of Background Radiation 86
5.4.1 Traffic Composition 87
5.4.2 Analysis of Backscatter Activity 89
5.5 Activities in Background Radiation 91
5.5.1 Details per Port 91
5.5.2 Temporal Distribution of Activities 99
5.6 Characteristics of Sources 102
5.6.1 Across Ports 102
5.6.2 Sources Seen Across Networks 104
5.6.3 Sources Seen Over Time 105
5.7 Summary 106

6 A First Look at Modern Enterprise Traffic 108

6.1 Problem Statement 109
6.2 Datasets 111

6.3 Broad Traffic Breakdown . . . 112
6.4 Origins and Locality . . . 117
6.5 Application Characteristics . . . 118
6.5.1 Internal/External Applications . . . 120
6.5.2 Enterprise-Only Applications . . . 130
6.6 Network Load . . . 139
6.7 Summary . . . 142

7 Conclusions 143

7.1 Contributions . . . 143
7.1.1 The `binpac` Language . . . 143
7.1.2 Semantic-Aware Traffic Anonymization . . . 144
7.1.3 Internet Background Radiation . . . 145
7.1.4 Enterprise Network Traffic . . . 145
7.2 Future Directions . . . 145
7.2.1 Traffic Analysis . . . 146
7.2.2 Traffic Anonymization . . . 148
7.3 Summary . . . 149

List of Figures

2.1 Declarations of `bro` events 14

2.2 Sample Bro script: (a simplified version of) Email relay detection 14

3.1 A HTTP parser in `binpac` with Bro event generation, complete except for MIME and escape-sequence processing (Part 1) 26

3.2 An (abridged) DNS parser in `binpac` (Part 1) 28

3.3 Specifying dynamic byte order with `&byteorder` 32

3.4 SUN/RPC per-connection state 35

3.5 Separating `bro` event generation from protocol syntax specification with `refine`. 37

3.6 Specifying buffering length of a type 38

4.1 Data Flow in Trace Transformation 48

4.2 Source and the Resulting C++ Code of a Rewrite Function 53

4.3 Deferring Writes to HTTP Content-Length Header 55

4.4 Anonymization Log Entries 66

4.5 FTP Reply Message Templates 67

4.6 Message Template Extraction 67

4.7 Execution time of various FTP policy scripts 70

5.1 Top level Umbrella of Application Responders 77

5.2 Example summary of port 445 activity on Class A. 506,892 sessions in total. Arcs indicate number of sessions. 79

5.3 Active response sequence for Samr-exe viruses 80

5.4 Effectiveness of Filtering, Networks (top) and Services (bottom) 83
5.5 The Honeynet architecture at iSink and LBL 85
5.6 Number of background radiation packets per hour seen at LBL 88
5.7 Time series of weekly backscatter in the four networks. 90
5.8 The components of backscatter at the class A network besides the dominant `RST`, `SYN-ACK`s. 90
5.9 Typical HTTP request of a tickerbar host 94
5.10 Port 135 activities on March 29 95
5.11 Unknown packets on UDP port 53 (DNS port) 97
5.12 Observed Windows Messenger Pop-Up Spam packets. 99
5.13 The Big Exploits (Apr 20 to May 7, 2004), as observed on 5 /C networks at LBL. The source hosts are counted every three hours. 100
5.14 Time series of activity on Agobot ports in the two UW /19 networks (on adjacent Class B networks) 102

6.1 Fraction of traffic payload bytes of various application layer protocols. 114
6.2 Fraction of packets of various application layer protocols. 115
6.3 Fraction of connections of various application layer protocols. 115
6.4 Locality in host communication. 119
6.5 HTTP fan-out. The N in the key is the number of samples throughout the chapter—in this case, the number of clients. 122
6.6 Size of HTTP reply, when present. 124
6.7 SMTP and IMAP/S connection durations. 126
6.8 SMTP and IMAP/S: flow size distribution 128
6.9 Windows (CIFS and DCE/RPC) Traffic Structure 131
6.10 NFS/NCP: number of requests per client-server pair, for those with at least one request seen. 135
6.11 NFS: request/reply data size distribution (message headers are not included) 137
6.12 NCP: request/reply data size distribution (message headers are not included) 138
6.13 Utilization distributions for D_4. 140
6.14 TCP retransmission rate across traces (for traces with at least 1000 packets in the category) 141

7.1 The spectrum of parallelism present in a high-level network security analysis pipeline. . . 147

List of Tables

3.1 Summary of `binpac` language constructs. 25

3.2 Number of lines of code (LOC), CPU time, and throughput of protocol analyzers 41

5.1 Summary of Data Collection 86

5.2 Protocol breakdown by packet rate. The rate is computed as number of packets per destination IP address per day, *i.e.*, with network size and sampling rate normalized 87

5.3 Protocol breakdown by number of sources. 87

5.4 The Most Popular TCP Ports. Ports that are visited by the most number of source IPs, as in a one week passive trace at LBL. In total there are 12,037,064 packets from 651,126 distinct source IP addresses. 88

5.5 Abbreviations for Popular Activities. Each line reflects a separate connection. 92

5.6 Port 80 Activities (March 29, 2004) Note that to reduce trace size the active responders at UW and Class A do not respond to `"SEARCH /"` to avoid getting the large `SrchAAA` requests. 93

5.7 Port 445 activities 95

5.8 Summary of DNS activity seen in the Class A (24 hours) 98

5.9 24 hours of multi-port source activity at the four sites 103

5.10 Traffic from sources seen across networks: intersections vs. individual networks 104

5.11 Traffic from sources seen over time: intersections vs. individual periods 105

6.1 Dataset characteristics. 111

6.2 Fraction of packets observed using the given network layer protocol. 112

6.3 Fractions of scan connections (removed from further analysis) 113

6.4 Fraction of connections and bytes utilizing various transport protocols. 113

6.5 Application categories and their constituent protocols. 114

6.6 Example application traffic characteristics. 120

6.7 Fraction of internal HTTP traffic from automated clients. 121

6.8 HTTP reply by content type. "Other" mainly includes *audio*, *video*, and *multipart*. 123

6.9 Email Traffic Size 124

6.10 Windows traffic connection success rate (by number of host-pairs, for internal traffic only) 131

6.11 CIFS command breakdown. "SMB basic" includes the common commands shared by all kinds of higher level applications: protocol negotiation, session setup/tear-down, tree connect/disconnect, and file/pipe open. 132

6.12 DCE/RPC function breakdown. 133

6.13 NFS/NCP Size 133

6.14 NFS requests breakdown. 134

6.15 NCP requests breakdown. 136

6.16 Backup Applications 136

Chapter 1

Introduction

An important aspect of building, managing, and improving a large and complex computer system is measuring its workload and behavior. This holds for computer networks, and is especially true for the Internet. This dissertation deals with one aspect of network measurements, *semantic traffic analysis*, which involves analyzing application layer payloads of network traffic to understand the behavior of network applications.

The first part of this dissertation presents techniques and tools that enable automated analysis of the application-level traffic data. On this foundation, the dissertation further develops a programming environment and techniques for packet trace anonymization that preserves application level data. Trace anonymization is critical to sharing traffic traces for network measurements and repeatable experiments.

Exploring opportunities brought by semantic traffic analysis tools, the second part of this dissertation presents measurements of two specific kinds of network traffic—Internet background radiation and enterprise internal traffic—that had not been studied previously. Both studies focus on the behavior of applications that are generating the traffic.

1.1 Background and Motivation

Network measurement can be roughly divided by which network layer it is concerned with. Most previous work has focused on lower layers and is generally concerned with understanding how bits move around the network, including: network topology, how many bits are moving from one end point to another (traffic matrix), what routes the bits are are taking (routing), and how to prevent congestion (congestion control).

Such studies are generally not concerned with the meaning of application layer traffic data and consider them as opaque bits.

This dissertation considers measurement at the application layer of the network: in particular, analyzing the contents of application level traffic payload. Through uncovering the meaning of application layer bits we can find out what applications are generating the traffic, for what functionality, whether the traffic is benign or malicious, how often application level failures occur, and so on. We refer to this general notion as *semantic traffic analysis*. It is worth noting that measurement of different network layers is closely related. Understanding application layer traffic semantics often helps reveal the underlying causes of lower-layer traffic patterns.

In practice, techniques developed for network traffic measurements can often be applied to real-time monitoring as well. Thus, understanding network traffic semantics also helps us to detect and thwart malicious activities (intrusion prevention), locate failures (trouble shooting), and project future usage patterns (provisioning).

Fundamentally, we measure network traffic semantics so as to understand the applications that are generating the traffic and, sometimes, the behavior of human beings using the applications. Thus, the ultimate target of measurement is often not the network itself in the narrow sense—links and routers—but rather the applications. Here we use the term *network application* in the broad sense—it includes not only the programs used directly by human users, such as Web browsers and email clients, but also any program that communicates over the network, such as clients and servers of DNS (Domain Name Service) and NTP (Network Time Protocol).

Interestingly, it is often easier to understand applications by measuring their network traffic than by analyzing the applications themselves. First, applications may not be directly accessible in measurements. For example, it is often infeasible to ask many users to install a Web browser extension for measuring Web browsing behaviors from the client side. In contrast, it is usually easier to capture and analyze the Web traffic on the network. Second, some applications do not have built-in mechanisms that record or report their activities—consider, for example, characterizing network activities generated by all applications running on a Windows desktop. Finally, even if applications do report their activities, for example, Web servers usually record requests in a log, the records often have formats particular to the applications, making it difficult to measure multiple applications with a unified mechanism. In comparison, the structure of network traffic is well-defined by the application protocols, and thus easier to parse. For all these reasons, network traffic provides a good window into the behavior of applications.

1.2 Challenges

Measuring a complex system is challenging in and of itself. This is especially true for large computer networks such as the Internet. Below is a short list of challenges network traffic measurement faces in general. We also discuss how the challenges impact measurement studies in this dissertation and tools and techniques we develop to overcome the difficulties.

Diversity of network applications. The Internet is a colorful world—it is used by hundreds of millions of users with thousands of applications. Moreover, the continual shift of our daily life towards computer networks ensures that the Internet diversity will continue to grow as we see ever more network applications. The large number of applications challenges us to build analysis tools for a wide range of the applications and protocols.

This dissertation tackles this challenge by presenting a language that facilitates the process of building analyzers for application protocols.

Irregularity in traffic data. Real-world traffic data contains various types of *irregularities*, ranging from deviation from standard protocol syntax, incorrect length fields or checksums, and completely corrupted data fields, to missing packets because of imperfect traffic capture mechanisms. Thus, measurement tools must be robust to handle such irregularity; measurement methodology should also take traffic irregularity into account.

Limited traffic data. The Internet consists of many autonomous and geographically distributed networks. It is often only possible to measure a very small part of it. Consider, for example, how many vantage points one needs to measure the total number of bytes (or HTTP requests) that fly over the Internet in a day. Note that the difficulty lies not only in the geographical distribution or number of systems, but also in the autonomy each system asserts. This latter factor is often more important, as most institutions are reluctant to reveal their network structure and traffic information because of concerns ranging from privacy and security to trade secrets. Because of the diversity of the Internet, lack of shared traffic data creates a significant hurdle for network measurements. For example, it is difficult to distinguish between universal and site-specific characteristics of network traffic based on traffic data from only one or two networks.

One key to creating shared traffic data is traffic anonymization—a process that removes sensitive data while preserving non-sensitive information which can be used in network measurements. This is another topic explored in this dissertation.

Asymmetric resource capacity between the measurement system and the system to measure. Even when one is measuring a very small part of the Internet, the network still usually contains far more computers than the measurement infrastructure does. As a result, much more computation resource is available for traffic generation than for traffic measurement. This resource asymmetry challenges us to design efficient measurement tools and sound sampling techniques, as explored in our study of Internet background radiation.

1.3 Dissertation Contributions

The dissertation makes two sets of contributions. The first involves developing techniques and tools for people to *analyze traffic semantics* and *anonymize traces*, motivated by the fact that lack of tools and traces has significantly hindered measurements of network traffic semantics. The second consists of measurements of two particular aspects of network traffic: *Internet background radiation* and *modern enterprise network traffic*. Both studies are the first characterizations of the respective type of traffic. Next we discuss each contribution in further details.

1.3.1 Tools and Techniques

To understand network traffic semantics we need to obtain traffic traces and build analysis tools. Below we discuss how this dissertation advances state of the art in both areas.

We note that the two problems are in fact closely related. An important channel to obtain network traces is through sharing anonymized traces. Trace anonymization, however, depends on semantic analysis tools that expose the semantics of data elements in the traffic, so that each element can be sanitized based on its application level meaning. On the other hand, in developing traffic analysis tools it is critical to have access to network traces so that the tools can be made robust enough to handle the irregularity found in real-world traffic.

A Declarative Language for Protocol Analyzers

A traffic analysis framework such as `bro` [89] processes network traffic in two stages (Chapter 2 offers a more detailed description of the process). In the first stage, the packets are translated into a high-level representation of the traffic, in case of `bro`, a sequence of application level *events*. For example, a `bro` *event* may represent an HTTP request, a DNS query, or begin or end of a connection. In the second stage, one can specify a rich set of analysis tasks that operate on the high level representation of the traffic. Under this model, deep traffic analysis can be expressed with a short script and carried out efficiently on high volume real-time traffic or large packet traces.

One of the main challenges in building such a network traffic analyzer such as `bro` is to parse a wide range of application protocols in order to generate the high-level events. Protocol parsers are usually built manually in a general-purpose programming languages such as C/C++ [29]. The construction and maintenance of analyzers requires a considerable amount of effort, because of the complexity of network protocols. For example, Netware Core Protocol has hundreds of message types for a wide range of functionalities from remote file access to printing and directory service. Furthermore, new application protocols are constantly emerging, so the analyzers must also be frequently updated. The question, naturally, is: How can we make it easier to build and maintain protocol parsers? The answer presented in this dissertation is a high-level, declarative language for specifying protocol parsers. The language's compiler generates efficient protocol parsers in C++ from the specification of (1) physical layout of protocol data, (2) semantic relations between protocol fields, and (3) custom computation (for example, `bro` event generation) to carry out during parsing. High-level specifications reduces complexity in two ways: (1) the compiler generates code for common and usually tedious tasks in traffic parsing, such as trans-coding byte order, checking boundaries, and buffering incomplete data, and thus reduces programming complexity and human errors; and (2) the separation of concerns—such as protocol syntax vs. `bro` event generation—makes the code easier to maintain and reuse. Use of this high-level language considerably reduces the complexity of building protocol parsers, and has helped us to quickly develop `bro` parsers for complicated protocols such as CIFS/SMB, DCE/RPC, and NCP in the study of enterprise traffic.

Semantic-aware Traffic Anonymization

Lack of packet traces has always been at the top of the list of impediments to network measurements [90, 5]. Lucky researchers may collect traces at their own institutions; unlucky ones have to live with nothing but synthetic traces. In an ideal world of network research, there would be packet traces of any type of traffic, with full payloads, collected on operational networks all over the world, and all available to the public. It would not only mitigate the problem of limited visibility in Internet measurements, but also allow comparing different techniques with the same data set and repeating experiments conducted by other researchers. A major reason we are not in such a world is that packet traces contain a lot of private information that cannot be released to the public. *Trace anonymization* is a process that removes sensitive information from traces in the hope that the remaining information is useful in measurements. Before the work presented in this dissertation, trace anonymization had been limited to TCP/IP headers, crippling research on Internet traffic semantics.

With the ability to understand network traffic at application semantic level, it is natural to go one step further—to build a programming environment for *transforming* network traffic, again, at application semantic level [80]. The goal is to allow a simple `bro` script to perform tasks such as replacing user ID "Alice" with "Bob", inserting new HTTP headers, and substituting Web items with their MD5 hashes. The underlying support environment makes corresponding changes at various protocol layers to keep the traffic well-formed. With the `bro` trace transformation framework, we solve two problems in trace anonymization: (1) how to find out what information is contained in the traces; (2) how to put transformed data back together as well-formed traces. We also investigate the remaining problem of how to obscure identities and other sensitive information through anonymizing FTP traces from the Lawrence Berkeley National Lab for public release, and in the process, developed methods to validate that anonymizations are both correctly specified and correctly applied.

1.3.2 Measurement Studies

Measurement of network traffic semantics is a largely unexplored area. Using the tools and techniques developed in this dissertation, we conducted first-time studies of two specific aspects of network traffic. As defined shortly below, *Internet background radiation* traffic is mostly generated by malicious programs (*malware*) that scan Internet addresses. Our study provides a broad view of ongoing activities of such

Internet malware, previously analyzed only on an individual basis. The enterprise traffic study, on the other hand, offers a broad look at the normal network traffic *within* an institution or corporation. It shows how the network traffic in a local, managed, and semi-closed environment differs from the well-studied wide-area Internet traffic.

Internet Background Radiation

Monitoring any portion of the Internet address space reveals incessant activity. This holds true even when monitoring traffic sent to unused addresses, which we term *background radiation*. While the general presence of background radiation is well known to the network operator community, the measurement presented in this dissertation is the first broad characterization of the radiation traffic [81]. One of our main objectives is to discover the *intentions* behind the packets sent to unused addresses. We build *network telescopes*—application responders on unused addresses to engage in conversation with the radiation sources, mimicking ordinary personal computers, till the intentions are revealed. As most radiation sources turn out to be Internet malware—worms and autorooters,[1] Internet background radiation offers a comprehensive and up-to-date view of Internet malware that scans the Internet for victims. We characterize the radiation traffic from various angles, for example, how many are active, how they probe and break into systems, and how they evolve over time.

Enterprise Network Traffic

Like background radiation, *enterprise network traffic* is also a previously unexplored realm. While many enterprise networks—institution/corporation internal networks—are connected to the Internet, activities occurring *within* internal networks usually do not appear in the wide-area traffic. Therefore, while the wide-area traffic had been well studied, little was known about the workload of enterprise internal networks. Because an enterprise network is managed by a single authority in a semi-closed environment, its workload may differ significantly from that of the wide-area network. The lack of measurement is a bit striking given the pressing problem of developing enterprise network management technologies. Modern enterprise networks have grown quite large and complex and become hard to manage. The largest enterprise networks today are larger than the entire Internet fifteen years ago—when the first measurements of the wide-area traffic were carried out. In this dissertation we

[1]"autorooter" is a jargon referring to software that automates the process of scanning and cracking systems to gain privileged access to computers.

present a first broad overview of modern enterprise traffic with traces from Lawrence Berkeley National Lab [77]. Our main goal is to provide a first sense of ways in which modern enterprise traffic is similar to wide-area Internet traffic, and ways in which it is quite different. We find enterprise internal traffic dominated by applications not commonly found in wide-area Internet traffic; and take a first look into these previously unstudied applications.

1.3.3 Thesis Outline

In summary, this dissertation advances our understanding of the meaning of Internet traffic by (1) building tools for developing application protocol parsers and traffic anonymizers, allowing wider, semantic access to Internet traffic beyond administrative limits; and (2) characterizing previously unstudied aspects of Internet traffic, including Internet background radiation and enterprise network traffic.

The rest of this dissertation is organized as follows. Chapter 2 gives an overview of the semantic structure of Internet traffic and the process of semantic traffic analysis. Chapter 3 then describes a high-level, declarative language for building a critical component of a traffic analysis platform—application protocol parsers. Chapter 4 presents a programming environment for packet trace transformation and anonymization. Chapter 5 and 6 present our characterization of Internet background radiation and modern enterprise traffic, respectively. This dissertation concludes in Chapter 7. As various areas of networking research overlap with different aspects of this dissertation, the related work is discussed separately in each Chapter.

Chapter 2

Traffic Semantic Analysis and `bro`

As many parts of this thesis are built around `bro`—a network traffic processing platform, this chapter presents an overview of the traffic semantic analysis process. It identifies where the challenges are, describes how they are handled by `bro`, and reports what open problems remain.

2.1 Traffic Capture

Measuring network traffic semantics starts with tapping network links to capture the traffic transmitted on the link. The captured traffic is usually in the form of a sequence of packets. Depending on the capture mechanism, the packets may contain various link layer headers. After stripping link layer headers, however, one can usually obtain packets in the Internet Protocol (IP) [93]. For simplicity, this dissertation assumes that a traffic analyzer can tap network links and observe all the IP packets flying by. [1]

2.2 Extracting High Level Traffic Representation

With traffic captured, the next step is to translate packets to a high-level representation of the traffic application behaviors. The translation process involves the following three steps.

[1] In practice, however, this assumption turns out to be a little over-optimistic. Sometimes the raw traffic contains too much sensitive information to be analyzed directly. Instead, the traffic data must be first "anonymized" to remove any sensitive information. In other cases high traffic volumes prohibit capture of full traffic and/or cause packet drops. People usually apply packet filters to capture traffic *selectively* and yet packet drops are still a common problem. However, for now suppose that the preceding assumption does hold.

2.2.1 TCP/IP Processing

The captured IP packets are the same as those seen by network routers. Each IP packet has a header, analogous to the envelope of a letter, with source and destination addresses and other relevant information for the packet to travel from the source host to the destination host. The format of IP packet payload—the contents of the "letter"—is defined by transport layer protocols.

If we consider packets between a pair of end hosts, the transport layer is responsible for carrying data between applications. Multiple applications on the same end host are indexed by *port* numbers; thus between the same pair of hosts there can be multiple channels, which are termed *connections*. There are two main transport layer protocols, TCP [94] and UDP [92]. Each TCP connection carries two streams of bytes, one on each direction, between applications; each UDP connection [2] carries two streams of datagrams. The structure of bytes and datagrams is determined by their respective applications.

As IP, TCP, and UDP are well known and simple protocols, it is relatively straightforward to extract what bytes or datagrams—which we term *application level data*—are sent between two applications with IP packets as input. To discover the meaning of these bytes, for example, whether the string "`root`" represents an administrator user ID or something else, one needs to determine (1) which applications are sending the bytes, and (2) how the bytes in the TCP byte streams or UDP datagrams are organized according to the specific application protocol. These are discussed below.

2.2.2 Recognizing Applications

The usual way to tell which applications are generating the traffic is by the port number of the server application, as ports are assigned to applications running on a host. (Note that we are interested in recognizing the general application category, such as Web vs. SMTP, instead of any particular software implementation, such as IIS vs. Apache Web server.)

Ports between 1 and 1024 are called "well-known ports" and are assigned to applications by IANA [46]. For instance, port 80/tcp is assigned to Web servers, and 53/udp for DNS servers. The assignment of well-known ports is a convention between clients and servers, so that the client applications, such as Web browsers, know to which port to send data for a particular service. There is nothing preventing one from running a mail server on the Web port, as long as the clients know to which port to connect.

[2]The UDP protocol is "connectionless", but we can consider each bidirectional pair of UDP datagram flows between the same pair of ports as a "connection".

There are also many applications that use ports above 1024. For example, the peer-to-peer application BitTorrent runs its “tracker” on port 6969. Some such ports are well-known, though they are not officially assigned.

Moreover, some applications do not run on any fixed port, but rely on some port mapping mechanism to direct clients to servers. For example, a client that wants to contact the Windows Messaging Service on another host first connects to the DCE/RPC port mapping server on port 135/tcp, which tells the client that the messaging service runs on port `X`; the client then connects to port `X` for the particular service. In such cases, one needs to parse the port mapping process to obtain the port-application mapping.

In summary, distinguishing applications by ports works well for a large fraction of network traffic (as evidenced by use of port-based firewalls as a basic tool for network defense). In some cases, it needs to be supplemented with semantic analysis of port mapping traffic. On the other hand, there is significant benefit in recognizing the part of traffic generated by well-known applications running on non-standard ports. Such traffic can potentially be more *interesting* because it might have been set up to evade detection and analysis. Ideally we would like to recognize applications by *contents* rather than by *convention* (well-known ports), Recent efforts by Dreger et al. [26] are an important step on this direction. In theory the problem is not completely solved because writing precise protocol recognizers requires manually deriving heuristics from protocol standards and tuning them with real traffic. However, in practice the heuristics tend to be quite robust and stable, and the problem is largely solved for well-known protocols.

2.2.3 Parsing Application Protocols

The second part of analyzing application level data involves dissecting byte streams and datagrams according to application protocols and generating high-level representations of the traffic. The task appears to be straightforward, yet in reality there are several major challenges.

First, there are a large number of application protocols. For instance, `Ethereal` [29], a network traffic monitor known for handling a wide range of application protocols, was able to parse 724 protocols as of November 2005. But the protocols handled by `Ethereal` represent only a small portion of all application protocols found in the Internet traffic [47].

Moreover, certain protocols are used for a wide range of purposes and thus are very complex. For example, Windows systems provide almost all their services, including file sharing, messaging, user authentication, and printing, through the combination of the CIFS protocol [19] and the DCE/RPC services.

There are 72 message types in the CIFS protocol, each type having several dozens of fields. The DCE/RPC protocol, including its various services, is even more complicated.

The number and complexity of protocols calls for a quick and reliable way to build protocol parsers.

Second, there is no public documentation for some protocols. Many protocols are standardized and well documented, but for some it is difficult to find any documentation in the public domain, either because the protocols are proprietary or just because the software vendor does not bother to make any documentation available. The vast number of protocols and lack of documentation makes it an intriguing research problem to reverse-engineer protocols automatically.

Third, the actual network traffic does not always conform to the protocols. Examples range from syntactic miscues, such as using a single `CR` or `LF`, instead of the standard `CR+LF`, as link breaks, to behavior deviations such as pipelining all SMTP commands, regardless whether they are pipelinable or not. To handle such deviations, the protocol parsers must be robust enough tolerate certain nonconformant behaviors and be able to recover from others.

Related to protocol irregularity is the problem of dealing with missing data. Traffic data can be incomplete due to packet drops in capturing network traffic or in observing long-standing connections that are established before traffic capture starts. A promising direction in handling incomplete traffic data is to build *re-synchronization* mechanisms into protocol analyzers, so that the analyzers can make good guesses as to where the messages are aligned and the state of the application session—for example, whether an SMTP connection is in command/reply conversation or in the middle of data transfer, and thereby recover from missing data.

Finally, there is the question of how to build protocol parsers *reusable* for a variety of analysis tasks with different requirements. To illustrate the problem, `Ethereal` provides a graphical user interface for people to investigate the traffic interactively, but it lacks support for automated traffic analysis. `Ethereal`'s parsers are difficult to reuse as they are tightly integrated with the rest of the system. Without parser reuse, it will take a significant amount of effort to build one's own parsers for each different type of analysis.

2.3 Traffic Analysis and Manipulation

With protocol parsers we are able to obtain a semantic level representation of the traffic. In the case of `bro`, the representation is in the form of one or multiple application level *events* for each packet. For example, a `bro` *event* may represent an HTTP request, a DNS query, or begin or end of a connection (Figure 2.1). The next question is how to utilize the high-level traffic representation and build specific traffic analyzers.

`bro`'s approach is to provide a *programming environment* for traffic analysis. Traffic analysis is programmed as a collection of *event* handlers in the `bro` scripting language. The `bro` language is Turing-complete, procedural, and strongly typed. It has built-in features commonly used in network traffic analysis, such as hash tables, regular expressions, and asynchronous events.

Consider, for example, building an Email relay detector that monitors traffic at the gateway of an enterprise network to detect Emails relayed through the enterprise network, based on the SMTP Message-ID headers [48]. Figure 2.2 shows a short script for detecting SMTP (Email) relays. It detects relays by looking for Email message ID's (as appear in the "MESSAGE-ID" header) that appear in multiple SMTP connections. The script maintains a mapping of message ID's to the first connections in which the ID's appears in hash table `msg_id_table`. (A table entry will expire and be evicted from the table if it is not accessed in a day.) The table is updated in event handler `mime_all_headers`.

`bro` generates a `mime_all_headers` event for each set of Email headers it encounters in an SMTP connection. The event carries two parameters—the connection in which the headers appear and the set of headers. Each header is represented by a *record* type with two members: `name` and `value`. The event handler in the script looks for headers of name "MESSAGE-ID" and looks up every message ID (found in the corresponding header value) in hash table `msg_id_table`. It inserts message ID's of first appearance to the table; for message ID's that have appeared before the script calls function `check_relay` for further processing.

This simple example highlights a few couple of key characteristics of traffic analysis in the `bro` programming environment. First, the analysis scripts operate on high-level data elements, such as names and values of Email headers, instead of packets or bytes. Second, the analysis can keep state across events and connections and may use data structures such as records and hash tables. Furthermore, the programming environment provides mechanisms to prevent accumulation of stale state—a common problem in network traffic analysis—by automatically expiring stale table entries.

The main challenge in building the `bro` programming environment is to understand requirements of traffic analysis and to design language features or run-time environment elements to meet the requirements. For example, hash tables are commonly used to keep state across events. For instance, the SMTP relay detection script keeps a table of seen message ID's. It is often necessary to make table entries expire over

```
event new_connection(c: connection);
event connection_finished(c: connection);

event http_reply(c: connection, version: string, code: count, reason: string);
event smtp_request(c: connection, is_orig: bool, cmd: string, arg: string);
```

Figure 2.1: Declarations of `bro` events

```
# A mapping from Email message ID to connection.
global msg_id_table: table[string] of connection &read_expire = 1 day;

event mime_all_headers(c: connection, hlist: mime_header_list)
    {
    for ( i in hlist )
        {
        local h = hlist[i];
        if ( h$name == "MESSAGE-ID" )
            {
            local msg_id = h$value;
            # Have we seen this message ID before?
            if ( msg_id !in msg_id_table )
                msg_id_table[msg_id] = c;
            else
                check_relay(msg_id, msg_id_table[msg_id], c);
            }
        }
    }
```

Figure 2.2: Sample Bro script: (a simplified version of) Email relay detection

time to limit memory consumption. `bro` scripts may specify a read or write expiration interval on a hash table and, optionally, a function to call when an entry expires.

Going one step further beyond traffic analysis, this dissertation also presents a framework for semantic-level traffic *manipulation*, built on top of the `bro` system. With the framework, a short `bro` script can insert or remove messages or headers, transform protocol fields, and write the resulting traffic data to a trace file. The trace will look as if the actual traffic was captured, i.e. the protocol fields (such as lengths and checksums) will remain consistent with each other after the transformation. Such a traffic manipulation mechanism provides a foundation for application-level trace anonymization (discussed further in Chapter 4).

2.4 Summary

With this overview we can see that traffic semantic analysis is a new field with many areas yet to be explored or advanced. The important open issues include recognizing applications by content, dissecting numerous and complex application protocols, recovering from errors, designing language features to facilitate analysis programming, and manipulating and anonymizing traffic.

Our main vehicle to explore these areas is Bro—a powerful traffic semantic analysis tool. In the process of building protocol analyzers for Bro, we also use Bro to investigate various kinds of network traffic and to anonymize traffic data. Consequently the dissertation aims to make contributions in multiple areas. It includes both techniques to facilitate analysis of traffic semantics and studies of Internet traffic at application semantic level.

Chapter 3

`binpac`—A `yacc` for Writing Application Protocol Parsers

A key step in the semantic analysis of network traffic is to parse the traffic stream according to the high-level protocols it contains. This process transforms raw bytes into structured, typed, and semantically meaningful data fields that provide a high-level representation of the traffic. However, constructing protocol parsers by hand is a tedious and error-prone affair due to the complexity and sheer number of application protocols.

This chapter presents `binpac`, a declarative language and compiler designed to simplify the task of constructing robust and efficient semantic analyzers for complex network protocols. It discusses the design of the `binpac` language and a range of issues in generating efficient parsers from high-level specifications. We have used `binpac` to build several protocol parsers for the "`bro`" network intrusion detection system, replacing some of its existing analyzers (handcrafted in C++), and supplementing its operation with analyzers for new protocols. We can then use `bro`'s powerful scripting language to express application-level analysis of network traffic in high-level terms that are both concise and expressive. `binpac` is now part of the open-source `bro` distribution.

The rest of this chapter is organized as follows. It begins with a discussion of background and motivation in Section 3.1, and related work in Section 3.2. Section 3.3 discusses specific characteristics of application protocols compared to languages targeted by traditional parser-generators. Section 3.4 describes the `binpac` language for specifying protocol syntax and the associated semantic analysis. Section 3.5

discusses the process of generating a parser from a `binpac` specification, including buffering of streaming input and performing robust error detection and recovery. Section 3.6 presents our experiences with using `binpac` to develop protocol parsers for the `bro` NIDS, and we compare their performance with that of manually written ones. Section 3.7 summarizes the chapter.

3.1 Problem Statement

Many network measurement studies involve analyzing network traffic in application-layer terms. For example, when studying Web traffic [7, 32] one often must parse HTTP headers to extract information about message length, content type, and caching behavior. Similarly, studies of Email traffic [52, 130], peer-to-peer applications [102], online gaming, and Internet attacks [81] require understanding application-level traffic semantics. However, it is tedious, error-prone, and sometimes prohibitively time-consuming to build application-level analysis tools from scratch, due to the complexity of dealing with low-level traffic data.

We can significantly simplify the process if we can leverage a common platform for various kinds of application-level traffic analysis. A key element of such a platform is *application-protocol parsers* that translate packet streams into high-level representations of the traffic, on top of which we can then use measurement scripts that manipulate semantically meaningful data elements such as HTTP content types or Email senders/recipients, instead of raw IP packets. Application-protocol parsers are also useful beyond network measurements—they form important components of network monitoring tools (e.g., tcpdump [49], Ethereal [29], NetDuDe [58]), real-time network intrusion detection systems (e.g., Snort [100, 101] and `bro` [89]), smart firewalls, and application layer proxies.

Building application-protocol parsers might appear straightforward at first glance, given a specification of the corresponding protocol. In practice, however, writing an efficient and robust parser is surprisingly difficult for a number of reasons. First, many of today's protocols are complex. For example, when analyzing the predominant HTTP protocol, one has to deal with pipelined requests, chunked data transfers, and MIME multipart bodies. The NetWare Core Protocol [95]—a common protocol used for remote file access—has about 400 individual request types, each with a distinct syntax. Second, even for simpler protocols, it is tedious and error-prone to manually write code to parse their structure: the code must handle thousands of connections in real-time to cope with the traffic in large networks, and protocol specifications are seldom comprehensive (i.e., they often ignore corner-cases, which a parser nevertheless must handle robustly as they *do* occur in real-world traffic). In potentially adversarial environments, an attacker may even deliberately craft ambiguous or non-conforming traffic [98, 43]. Furthermore, several severe vulnerabilities have been discovered in existing protocol parsers ([119, 120, 121, 122])—including one which enabled a worm to propagate through 12,000 deployments of a security product worldwide in tens of minutes [108, 60]—which demonstrates how difficult it is to comprehensively accommodate non-conforming input with hand-written code.

Given the care that writing a good protocol analyzer requires, it is unfortunate that existing analyzers are generally not reusable, because their operation is usually tightly coupled with their specific application environments. For instance, the two major open-source network intrusion detection systems (NIDSs), Snort [101] and `bro` [89], both provide their own HTTP analyzers, each exhibiting different features and shortcomings. Ethereal contains a huge collection of protocol parsers, but it is very difficult to reuse them for, e.g., `bro` due to their quite different interfaces and data structures. Even inside a single software product, low-level code is generally inlined rather than factored into modules. For example, the Ethereal version 0.99 source code contains more than 8,000 instances of incrementing or decrementing by a hard-coded numeric constant, the vast majority of which are adjusting a pointer or a length while stepping through a buffer. Any instance of an incorrect constant can of course result in an incorrect parsing of a protocol, but the mistake would not be detectable at compile-time since using the wrong numeric constant still type-checks.

We believe that the major reason for all of these difficulties is a significant lack of abstraction. In the programming language community, no one writes parsers manually. Instead, there are tools like `yacc` and ANTLR [83] that support *declarative* programming: one expresses the syntax of the language of interest in a high-level meta-grammar, along with associated semantics. The parser generator then translates the specification into low-level code automatically. In this work, we propose to use similar abstractions for application-layer network protocols. By doing so, users building analyzers can concentrate on high-level protocol semantics, while at the same time achieving correctness, robustness, efficiency, and reusability.

However, we observe that existing parser-generation tools are not suitable for parsing network protocols. Common idioms of network protocols, such as data fields preceded by their actual length (sometimes not adjacent), cannot be easily expressed as a context-free grammar. Furthermore, when analyzing protocols, we often need to correlate across the two directions of a single connection; sometimes even syntax depends on the semantics of the byte stream in the other direction. Finally, parsers generated by these tools process input in a "pull" mode and thus cannot concurrently parse multiple, incomplete input streams.

To improve this situation we designed and implemented `binpac`—a declarative language and its compiler—to simplify the task of building protocol analyzers. Users specify parsers by defining message formats, dependencies between message fields, and additional computations to perform (e.g., printing ASCII records or triggering further analysis) upon parsing different message elements. The compiler translates the declarations into parsers in C++. `binpac` takes care of all the common and tedious (and thus error-prone) low-level tasks, such as byte-order handling, application-layer fragment reassembly, incremental input, boundary checking, and support for debugging. `binpac` also facilitates protocol parser *reuse* by supporting separation of different components of analyzers. One can readily plug in or remove one part of a protocol analyzer without modifying others. Such separation allows analysis-independent protocol specifications to be reused by different analysis tasks, and simplifies the task of protocol extension (for example, adding or removing NFS to the RPC parser).

Our goal is to ensure that the generated parsers are as efficient as carefully hand-written ones, so that they can handle large traffic volumes. Our main strategy is to leverage *data dependency analysis*—to tailor the generated parser to the analysis requirements at *compilation time*. For example, `binpac` identifies appropriate units for buffering of incomplete input based on the data layout specified by the user.

To demonstrate the power of our approach, we have used `binpac` to build several protocol parsers for the `bro` NIDS, including HTTP, DNS, CIFS/SMB, DCE/RPC, NCP, and Sun RPC. (We emphasize that `binpac` is not however tied in any significant way to `bro`.) Having written many protocol analyzers manually in the past, our experience is that `binpac` greatly eases the process. In future work we envision further using these `binpac` specifications to compile analyzers to alternative execution models: in particular, directly to custom hardware, without any intermediate C++ code, as sketched in [85].

3.2 Related Work

Considerable previous work has addressed facets of describing data and protocol layouts using declarative languages. First, there are various Interface Description Languages for describing the service interface for specific protocols. For instance, the External Data Representation Standard (XDR) [111] defines the way to describe procedure parameters and return values for the Remote Procedure Call (RPC) protocol [110].

The XDR compiler generates the underlying code to marshall/unmarshall data to/from raw bytes. Targeting a wider range of protocols, ASN.1 [8] is a language for describing the abstract syntax of communication protocols, including a set of encoding schemes. Unlike `binpac`, these languages dictate the underlying data representation, while `binpac` tries to describe the data layout of a wide range of existing (thus, already designed) protocols that span a variety of formats and styles.

Augmented BNF (ABNF) [20] is used in many protocol standards to specify the protocol syntax. However, the goal of ABNF is to provide a concise, yet incomplete, way to define a protocol, rather than for complete protocol specification from which one can generate a parser. In addition, ABNF targets ASCII protocols.

People have also designed languages for writing network protocol implementations, including both protocol parsing and processing logic. Abbott et al. [1] proposed a language for designing and implementing new network protocols. Prolac [56] is a language for writing modular implementations of networking protocols such as TCP. Biagioni et al. [14] experimented with implementing a TCP/IP network stack in ML. These efforts differ from `binpac` in that the goal is to build end-system implementations, instead of analyzers, of protocols. They also target protocols at the network and transport layers, rather than the wide range of application protocols.

More related to `binpac`, there are efforts in the abstract description of *existing* protocol syntax. McCann and Chandra introduced PACKETTYPES [65], a language that helps programmers to handle binary data structures in network packets as if they were C types. Borisov et al. designed and implemented GAPA [15], a framework for application protocol analyzers. Its protocol specification language, GAPAL, is based on (augmented) BNF, but supports both ASCII and binary protocols. A protocol specification in GAPAL includes both protocol syntax as well as analysis state and logic. While GAPA and `binpac` both target application-level traffic analysis in general, they are designed with different sets of goals and therefore take quite different approaches. First, GAPA targets traffic analysis at individual end hosts, and uses an interpreted, type-safe language. The `binpac` compiler, on the other hand, generates C++ parsers intended to process traffic of much higher volume at network gateways. Second, GAPA is a self-contained system that handles both protocol parsing and traffic analysis with a single script. `binpac` is designed as a building block for the construction of parsers that can be used by separate traffic analysis systems such as `bro`, and the `binpac` language stresses separation of parsing and analysis.

Beyond network protocols, there are a number of languages for describing data formats in general. DATASCRIPT [10] is a scripting language with support for describing and parsing binary data. Developed more recently, PADS is a language for describing ad hoc data formats [34]. PADS's approach to data layout description is similar to that of `binpac` in a number of ways, such as the use of parameterized types. On the other hand, it is designed for a more general purpose than parsing network protocols, so it lacks abstractions and features particular to processing communication traffic, and the generated parsers cannot handle many input streams simultaneously. Related to PADS, Fisher et al. [35] described a calculus for reasoning about properties and features of data description languages in general. The calculus is used to discover subtle bugs, to prove the type correctness of PADS, and to guide the design of language features.

Hand-written application-layer protocol parsers are an important part of many network analysis tools. Packet monitors such as tcpdump [49], Ethereal [29], and DSniff [27] display protocol information. NetDuDe [58] provides both visualization and editing of packet traces. NIDS such as Snort [101], `bro` [89], and Network Flight Recorder [75] analyze protocol communications to detect malicious behavior. Protocol parsers are also components of smart firewalls and application-layer proxies.

3.3 Characteristics of Application Protocols

In this section we examine characteristics of network protocols which differ significantly from the sorts of languages targeted by traditional parser-generators. We discuss them in terms of syntax and grammar, input model, and robustness.

3.3.1 Syntax and Grammar Issues

In terms of syntax and grammar, application-layer protocols can be broadly categorized into two classes: *binary* protocols and human-readable *ASCII* protocols. The messages of binary protocols, like DNS and CIFS, consist of a (not necessarily fixed) number of data *fields*. These fields directly map to a set of basic data *types* such as integers and strings. Clear-text ASCII protocols, on the other hand, typically restrict their payload to a human-readable request/reply structure, using only printable ASCII-characters. Many of these protocols, such as HTTP and SMTP, are primarily line-based, i.e., requests/replies are separated by carriage-return/line-feed (CR/LF) tuples, and their syntax is usually specified with grammar production rules in protocol standards.

While these two types of protocols appear to exhibit quite distinct language characteristics, we in fact find enough underlying commonality between binary and ASCII protocols that we can treat both styles in a uniform fashion within declarative `binpac` specifications, as we will develop below. On the other hand, there are some critical differences between the grammars of network protocols (binary as well as ASCII) and those of programming languages:

Variable-length arrays. A common pattern in protocol syntax is to use one field to indicate the length of a later array. Such a length field often delimits the length of a subsequent (not necessarily contiguous) byte sequence, e.g., the HTTP "Content-Length" field, but can also indicate the number of complex elements, such as in the case of DNS question and answer arrays. A conceptual variant of variable-length arrays is *padding*, i.e., filling a field with additional bytes to reach a specific length.

As long as the length-field has constant width, it is theoretically possible to describe arrays and padding with a context-free grammar. However, doing so is cumbersome and leads to complex grammars.

Selecting among grammar production rules. Both binary and ASCII protocols often use one or multiple data fields to select the interpretation of a subsequent element from a range of options. For example, DNS uses a type field to differentiate between various kinds of "resource records". HTTP uses multiple header fields to determine whether the message body is a consecutive byte sequence, a sequence of byte chunks, or multipart entities. Sometimes the selector even comes from the opposite flow of the connection, e.g., the syntax of a SUN/RPC reply depends on program and procedure fields in the corresponding RPC call [110]. In general such a selector can be easily expressed in a grammar by *parameterizing* non-terminal symbols—a very limited form of context-sensitive grammar which we describe later in Section 3.4.1, However it is very hard to specify a selector with a *context-free* grammar.

Encoding. In binary protocols, record fields directly correspond to values. Therefore it is crucial to consider the correct byte-encoding when parsing fields. For example, integers are often either encoded in big-endian or little-endian byte order. Similarly, string characters may be given in a (single-byte) ASCII encoding or in a (two-byte) Unicode representation. To complicate the problem, the byte order need not be fixed for a given protocol. For example, there is a field in DCE/RPC header which explicitly indicates the byte order in which subsequent integers are encoded. In CIFS, a similar field gives the character encoding for strings (which in fact does not apply to all strings: certain ones are always in ASCII; similarly in CIFS, not all integers use the same byte-order). Handling data encoding is a tedious and error-prone task when writing a parser manually, and it is hardly expressible by means of an LALR(1) grammar [1].

3.3.2 Concurrent Input

A fundamental difference between a protocol parser and a `yacc`-style parser is their input model. A protocol-parser has to parse many connections simultaneously and, within each connection, the two flows on opposite directions, *in parallel*. For example, in persistent HTTP connections each request needs to be associated with the correct reply. Similarly, the syntax of a SUN/RPC reply depends on program and procedure fields in the corresponding RPC call [110].

Parsers generated by `yacc/lex` process input in a "pull" fashion. That is, when input is incomplete, the parser blocks, waiting for further input. Thus, a thread can handle only one input stream at a time. To handle flows simultaneously without spawning a thread for each one, the parsers must instead process input *incrementally* as it comes in, partially scanning and parsing incomplete input and resuming where the analysis left off when next invoked.

3.3.3 Robustness

Parsing errors are inevitable when processing network traffic. Errors can be caused by irregularity in real-world traffic data (protocol deviations, corrupted contents) as well as by incomplete input due to packet drops when capturing network traffic, asymmetric routing (so that only one direction of a connection is captured), routing changes, or "cold start" (a connection was already underway when the monitor begins operation). Unlike compilers, protocol parsers cannot simply complain and stop processing, but must robustly detect and recover from errors. This is particularly important if we consider the presence of adversaries: an attacker might specially craft traffic to lead a protocol parser into an error condition.

[1]LALR(1) grammar stands for Look-Ahead LR(1) grammar—a grammar family accepted by common parser generators such as `yacc`. Please see [3] for a detailed discussion of LALR and LR grammars.

3.4 The `binpac` Language

In the previous section we examined the grammatical characteristics of network protocols. This section describes the design of the `binpac` language and its compiler, which are specifically tailored to address these properties.

We begin with a description of `binpac`'s *data model* in Section 3.4.1, corresponding to production rules in BNF grammars. In Section 3.4.2 we discuss state-holding, in Section 3.4.3 how to add custom computation, and finally in Section 3.4.4 the "separation of concerns" to provide reusability.

Throughout the discussion we will refer to the examples in Figures 3.1 and 3.2, which show specifications of HTTP and DNS parsers in `binpac`, respectively. We use them to illustrate features of the `binpac` language. Note that the HTTP parser shown in Figure 3.1 is *complete* by itself (though simplified from the fully-featured one we built for `bro`, and evaluate below), except for the MIME-decoding of HTTP bodies and escape sequences for URIs. The former takes significant additional work to add; the latter can be incorporated easily by processing the raw, extracted URI with an additional function call. Due to space limitations, we only show an excerpt of the DNS parser, though this includes the technically most difficult element of parsing the protocol, namely compression-by-indirection of domain names.

In the language, text between %.*{ and %} embeds C/C++ code. `binpac` keywords reflecting optional attributes start with "&" (e.g., `&oneline`). Keywords starting with "$", such as `$context` and `$element`, are macros instantiated during parsing. In the examples, we highlight `binpac` keywords (except for elementary types, introduced below) using bold slant fonts. Table 3.4 summarizes the `binpac` language constructs.

3.4.1 Data Model

`binpac`'s data model provides integral and composite types which allow us to describe basic patterns in protocol data layout, parameterized types to pass information between grammar elements, and derivative data fields to store intermediate computation results. We discuss these in turn.

Language Construct	Brief Explanation	Section	Example
`%header{ ... %}`	Copy the C++ code to the generated header file		Fig. 3.1, #15
`%code{ ... %}`	Copy C++ code to the generated source file		
`%member{ ... %}`	C++ declarations of private class members of connection or flow	§3.4.2	Figure 3.4
`analyzer ... withcontext`	Declare the beginning of a parser module and the members of `$context`	§3.4.2	Fig. 3.1, #1
`connection`	Define a connection object	§3.4.2	Fig. 3.1, #37
`upflow/downflow`	Declare flow names for two flows of the connection	§3.4.2	Fig. 3.1, #38
`flow`	Define a flow object	§3.4.2	Fig. 3.1, #40
`datagram = ... withcontext`	Declare the datagram flow unit type	§3.4.2	Fig. 3.2, #64
`flowunit = ... withcontext`	Declare the byte-stream flow unit type	§3.4.2	Fig. 3.1, #41
`enum`	Define a “enum” type		Fig. 3.1, #5
`type ... =`	Define a `binpac` type	§3.4.1	Fig. 3.1, #11
`record`	Record type	§3.4.1	Fig. 3.1, #49
`case ... of`	Case type: representing an alternation among case field types	§3.4.1	Fig. 3.1, #45
`default`	The default case	§3.4.1	Fig. 3.1, #103
⟨`type`⟩`[]`	Array type	§3.4.1	Fig. 3.1, #87
`RE/.../`	A string matching the given regular expression	§3.4.1	Fig. 3.1, #11
`bytestring`	An arbitrary-content byte string	§3.4.1	Fig. 3.1, #73
`extern type`	Declare an external type	§3.4.1	Fig. 3.1, #13
`function`	Define a function	§3.4.2	Fig. 3.2, #67
`refine typeattr`	Add a type attribute to the `binpac` type	§3.4.4	Fig. 3.5
⟨`type`⟩ `withinput` ⟨`input`⟩	Parse ⟨type⟩ on the given ⟨input⟩ instead of the default input	§3.4.1	Fig. 3.2, #59
`&byteorder`	Define the byte order of the type and all enclosed types (unless otherwise specified)	§3.4.1	Fig. 3.2, #7
`&check`	Check a predicate condition and raise an exception if the condition evaluates to false	§3.5.2	Fig. 3.2, #34
`&chunked`	Do not buffer contents of the bytestring, instead, deliver each chunk as `$chunk` to `&processchunk` (if any is specified)	§3.4.1	Fig. 3.1, #100
`&exportsourcedata`	Makes the source data for the type visible through a member variable `sourcedata`	§3.4.1	Fig. 3.2, #7
`&if`	Evaluate a field only if the condition is true		Fig. 3.2, #16
`&length = ...`	Length of source data should be ...	§3.4.1	Fig. 3.1, #101
`&let`	Define derivative types	§3.4.1	Fig. 3.1, #63
`&oneline`	Length of source data is one line	§3.5.1	Fig. 3.1, #63
`&processchunk`	Computation for each `$chunk` of bytestring defined with `&chunked`	§3.4.1	
`&requires`	Introduce artificial data dependency		
`&restofdata`	Length of source data is till the end of input	§3.4.1	Fig. 3.1, #73
`&transient`	Do not create a copy of the bytestring	§3.6	
`&until`	End of an array if condition (on `$element` or `$input`) is satisfied	§3.4.1	Fig. 3.1, #87

Table 3.1: Summary of `binpac` language constructs.

```
analyzer HTTP withcontext {  # members of $context
    connection: HTTP_Conn;
    flow:       HTTP_Flow;
};
enum DeliveryMode {
    UNKNOWN_DELIVERY_MODE,
    CONTENT_LENGTH,
    CHUNKED,
};
# Regular expression patterns
type HTTP_TOKEN = RE/[^()<>@,;:\\"\/\[\]?={} \t]+/;
type HTTP_WS    = RE/[ \t]*/;
extern type BroConn;
extern type HTTP_HeaderInfo;
%header{
    // Between %.*{ and %} is embedded C++ header/code
    class HTTP_HeaderInfo {
    public:
        HTTP_HeaderInfo(HTTP_Headers *headers) {
          delivery_mode = UNKNOWN_DELIVERY_MODE;
          for (int i = 0; i < headers->length(); ++i) {
            HTTP_Header *h = (*headers)[i];
            if (h->name() ==[2] "CONTENT-LENGTH") {
              delivery_mode = CONTENT_LENGTH;
              content_length = to_int(h->value());
            } else if (h->name() == "TRANSFER-ENCODING"
                  && has_prefix(h->value(), "CHUNKED")) {
              delivery_mode = CHUNKED;
            }
          }
        }
        DeliveryMode delivery_mode;
        int content_length;
    };
%}
# Connection and flow
connection HTTP_Conn(bro_conn: BroConn) {
    upflow = HTTP_Flow(true);  downflow = HTTP_Flow(false);
};
flow HTTP_Flow(is_orig: bool) {
    flowunit = HTTP_PDU(is_orig)
                     withcontext(connection, this);
};
# Types
type HTTP_PDU(is_orig: bool) = case is_orig of {
    true  -> request: HTTP_Request;
    false -> reply:   HTTP_Reply;
};
type HTTP_Request = record {
    request:    HTTP_RequestLine;
    msg:        HTTP_Message;
};
type HTTP_Reply = record {
    reply:      HTTP_ReplyLine;
    msg:        HTTP_Message;
};
```

Figure 3.1: A HTTP parser in `binpac` with Bro event generation, complete except for MIME and escape-sequence processing (Part 1)

```
type HTTP_RequestLine = record {
    method:     HTTP_TOKEN;
    :           HTTP_WS;    # an anonymous field has no name
    uri:        RE/[[:alnum:][:punct:]]+/;
    :           HTTP_WS;
    version:    HTTP_Version;
} &oneline, &let {
    bro_gen_req: bool = bro_event_http_request(
        $context.connection.bro_conn,
        method, uri, version.vers_str);
};
type HTTP_ReplyLine = record {
    version:    HTTP_Version;
    :           HTTP_WS;
    status:     RE/[0-9]\{3\}/;
    :           HTTP_WS;
    reason:     bytestring &restofdata;
} &oneline, &let {
    bro_gen_resp: bool = bro_event_http_reply(
        $context.connection.bro_conn,
        version.vers_str, to_int(status), reason);
};
type HTTP_Version = record {
    :           "HTTP/";
    vers_str:   RE/[0-9]+\.[0-9]+/;
};
type HTTP_Message = record {
    headers:    HTTP_Headers;
    body:       HTTP_Body(HTTP_HeaderInfo(headers));
};
type HTTP_Headers = HTTP_Header[] &until($input.length() == 0);
type HTTP_Header = record {
    name:       HTTP_TOKEN;
    :           ":";
    :           HTTP_WS;
    value:      bytestring &restofdata;
} &oneline, &let {
    bro_gen_hdr: bool = bro_event_http_header(
        $context.connection.bro_conn,
        $context.flow.is_orig, name, value);
};
type HTTP_Body(hdrinfo: HTTP_HeaderInfo) =
           case hdrinfo.delivery_mode of {
    CONTENT_LENGTH -> body: bytestring &chunked,
                           &length = hdrinfo.content_length;
    CHUNKED        -> chunks: HTTP_Chunks;
    default        -> other: HTTP_UnknownBody;
};
type HTTP_Chunks = record {
    chunks:     HTTP_Chunk[] &until($element.chunk_length == 0);
    headers:    HTTP_Headers;
};
type HTTP_Chunk = record {
    len_line:   bytestring &oneline;
    data:       bytestring &chunked, &length = chunk_length;
    opt_crlf:   case chunk_length of {
        0       -> none: empty;
        default -> crlf: bytestring &oneline;
    };
} &let {
    chunk_length: int = to_int(len_line, 16);  # in hexadecimal
};
```

Figure 3.1: A HTTP parser in `binpac` with Bro event generation, complete except for MIME and escape-sequence processing (Part 2)

```
type DNS_message = record {
   header:     DNS_header;
   question:   DNS_question(this)[header.qdcount];
   answer:     DNS_rr(this)[header.ancount];
   authority:  DNS_rr(this)[header.nscount];
   additional: DNS_rr(this)[header.arcount];
} &byteorder = bigendian, &exportsourcedata;
type DNS_header = record { ... };
type DNS_question(msg: DNS_message) = record {
   qname: DNS_name(msg);  qtype: uint16;  qclass: uint16;
} &let {
   # Generate bro event dns_request if a query
   bro_gen_request: bool = bro_event_dns_request(
           $context.connection.bro_conn,
           msg.header, qname, qtype, qclass)
       &if (msg.header.qr == 0); # if a request
};
type DNS_rr(msg: DNS_message) = record {
   rr_name:    DNS_name(msg);
   rr_type:    uint16;   rr_class:   uint16;
   rr_ttl:     uint32;   rr_rdlen:   uint16;
   rr_rdata:   DNS_rdata(msg, rr_type, rr_class)
               &length = rr_rdlen;
} &let {
   bro_gen_A_reply: bool  = bro_event_dns_A_reply(
           $context.connection.bro_conn,
           msg.header, this, rr_rdata.type_a)
       &if (rr_type == 1);
   bro_gen_NS_reply: bool = bro_event_dns_NS_reply(...);
       &if (rr_type == 2);
};
type DNS_rdata(msg: DNS_message, rr_type: uint16,
       rr_class: uint16) = case rr_type of {
   1   -> type_a:   uint32 &check(rr_class == CLASS_IN);
   2   -> type_ns:  DNS_name(msg);
   # Omitted: TYPE_PTR, TYPE_MX, ...
   default -> unknown:  bytestring &restofdata;
};
```

Figure 3.2: An (abridged) DNS parser in binpac (Part 1)

```
# A DNS name is a sequence of DNS labels
type DNS_name(msg: DNS_message) = record {
   labels:     DNS_label(msg)[] &until($element.last);
};

# A label contains a byte string or a name pointer
type DNS_label(msg: DNS_message) = record {
   length:     uint8;
   data:       case label_type of {
       0 ->    label:  bytestring &length = length;
       3 ->    ptr_lo: uint8;   # the lower 8-bit of offset
   };
} &let {
   label_type: uint8   = length >> 6;
   last: bool          = (length == 0) || (label_type == 3);

   # If the label is a pointer ...
   ptr_offset: uint16 = (length & 0x3f) << 8 + ptr_lo
       &if(label_type == 3);
   ptr: DNS_name(msg)
       withinput msgdata(msg.sourcedata, ptr_offset)
       &if(label_type == 3);
};

flow DNS_Flow {
  datagram = DNS_message withcontext (connection, this);

  # Returns the byte segment starting at <offset> of <msgdata>
  function msgdata(msgdata: const_bytestring,
                         offset: int): const_bytestring
     %{
     // Omitted: DNS pointer loop detection
     if ( offset < 0 || offset >= msgdata.length() )
       return const_bytestring(0, 0);
     return const_bytestring(msgdata.begin() + offset,
                             msgdata.end());
     %}
};
```

Figure 3.2: An (abridged) DNS parser in `binpac` (Part 2)

Integral and Composite Types

A `binpac` *type* describes both the data layout of a consecutive segment of bytes and the resulting data structure after parsing. Type `empty` represents zero-length input. Elementary types `int8`, `int16`, `int32` represent 8-, 16-, and 32-bit integers, respectively, and so do their unsigned counterparts, `uint8`, `uint16`, and `uint32`. As the specification of `HTTP_ReplyLine` shows (Figure 3.1), a string type can be represented with a constant string (line 80), a regular expression (line 81), or a generic `bytestring` either of a specific length (with `&length`, line 101) or running till the end of data (with `&restofdata`, line 92).

Elementary integer and string types map naturally to their counterparts in C++ (in the case of string, we define a simple C++ class to denote the begin and end of the string). This is how the results are stored and accessed, with one exception. We allow a string to be "*chunked*" to handle potentially very long byte sequences, such as HTTP bodies, with a `&chunked` attribute (Figure 3.1, line 100). A chunked string is not buffered. Rather, with the `&processchunk` attribute one may define computation on each chunk to process the byte sequence in a streaming fashion. For instance, to compute a MD5 checksum for every HTTP body we may add a `&processchunk` as follows (assuming `compute_md5` maintains intermediate results across chunks):

```
http_body: bytestring &chunked,
      &length = $context.flow.content_length(),
      &processchunk($context.flow.compute_md5($chunk));
```

External C++ types, including `bool`, `int`, and user-defined ones (declared with `extern type`), can be used in computation, e.g., as types of parameters, but cannot appear as types of data fields in protocol messages.

Users can define composite types: (1) `record`, a sequential collection of fields of different types; (2) `case`, a union (in the C-language sense) of different types; and (3) `array`, a sequence of single-type elements. `binpac` generates a C++ class for each user-defined type, with data fields mapped as class members, and a parse function to process a segment of bytes to extract various data fields according to the layout specification.

As we compare `record` and `case` types with context-free grammar production rules, we can see a clear correspondence between them: a concatenation of symbols maps to a `record` type and multiple production rules of a symbol map to a `case` type. But there is a difference in the latter mapping. In the case of LR grammars, the choice of which production rules to follow is determined by looking ahead one or more symbols. In contrast, the `case` type corresponds to a set of production rules with *zero look-ahead*. Instead, a production rule is selected based on values of earlier fields—for example, DNS record type determines how to parse the record contents—and, more generally, an explicit indexing expression computed from other data fields or type parameters (Figure 3.1, line 99). This allows production rule selection to be based on external information, and is in spirit similar to "predicated parsing" introduced in ANTLR [83]. On the other hand, the zero-look-ahead restriction simplifies parser construction, but at the same time poses little limitation on the range of protocols that can be specified in `binpac`. The reason that there are few protocol syntax patterns that require look-ahead, we believe, is by design of protocols. Since protocol data is generated and processed by programs, it is usually organized in a way that simplifies the (traditionally hand-written) implementation.

Although an array can be defined with recursive production rules, we find it a common enough idiom in protocol syntax that it justifies a separate abstraction. In `binpac`, the length of an array can be specified with an expression containing references to other data fields, as in the definition of `DNS_message` (Figure 3.2, line 3-6). An array can also be defined without a length, but with some "terminate condition" that indicates the end of array. Such a condition is specified through the `&until` attribute with a conditional expression. The expression can be computed from the input data to each element (`$input`), as in `HTTP_Headers` (Figure 3.1, line 87), or from a parsed element (`$element`), as in `HTTP_Chunks` (line 106).

Type Parameters

As the examples of HTTP and DNS parsers show, *type parameters* (e.g., in type `HTTP_Body`, Figure 3.1, line 98) allow one to pass information between types without resorting to keeping external state. This is a powerful feature that can significantly simplify syntax specification.

```
type NDR_Format = record {
    # Note, field names taken from DCE/RPC spec.
    intchar   : uint8;
    floatspec : uint8;
    reserved  : padding[2];
} &let {
    ndr_byteorder = (intchar & 0xf0) ?
             littleendian : bigendian;
};

type DCE_RPC_Message = record {
    # Raise an exception if RPC version != 5
    rpc_vers       : uint8 &check(rpc_vers == 5);
    rpc_vers_minor : uint8;
    PTYPE          : uint8;
    pfc_flags      : uint8;
    # ‘drep’--data representation
    packed_drep    : NDR_Format;
    ...
} &byteorder = packed_drep.ndr_byteorder;
```

Figure 3.3: Specifying dynamic byte order with `&byteorder`

Byte Orders

For use with binary protocols, `binpac` allows the user to specify the byte order using a `&byteorder` attribute. Figure 3.3 shows the specification of dynamic byte-order in DCE/RPC, where at the bottom the user specifies that the byte-order is taken from the `ndr_byteorder` field that is defined earlier.[3]

In most cases we also want to propagate the byte-order specification along the type hierarchy to the other types. Conceptually we can pass byte order between types as a parameter (see Section 3.4.1), but in practice the byte order parameter is required universally for binary protocols. Adding a parameter to each type would be tedious and clutter the specification. To solve this problem, we designate “byteorder” as an *implicit type parameter* that is always passed to referenced types unless it is redefined at the referenced type. The `binpac` compiler traverses the type reference graph to find out which types require byte-order specification and adds byte order parameters to their parse functions.

We have not yet added support for ASCII vs. Unicode to `binpac`, though conceptually it will be similar to the support for byte-order.

[3]We discuss the definition of “derivative fields” such as `ndr_byteorder` in Section 3.4.1.

Derivative Fields

Sometimes it is useful to add user-defined *derivative fields* to a type definition to keep intermediate computation results (see the definition of `HTTP_Chunk.chunk_length` in Figure 3.1, line 117), or to further process parsing results (`DNS_label` in Figure 3.2, line 58-60). Derivative fields are specified within `&let {...}` attributes.

A derivative field may take one of two forms. First, a derivative field can be defined with an expression, in the form of "`<id> = <expression>`", as in the HTTP example.

Second, it can be evaluated by mapping a type onto a piece of computed input, in the form of "`<id>: <type>` ***`withinput`*** `<input expression>`". Here `<input expression>` evaluates to a sequence of bytes, which are passed to the parse function of `<type>` as input data. Such `withinput` fields allow us to extend parsing beyond consecutive and non-overlapping pieces of original input data. For instance, the computed input data might be (1) a reassembly of fragments (e.g. a fragmented DCE/RPC message body), (2) a Base64-decoded Email body, or (3) a DNS name pointer, as shown in Figure 3.2, line 55-60. In the DNS example, a DNS label can be a sequence of bytes or a "name pointer" pointing to a DNS name at some specific offset of the message's source data. In the latter case, we define a `withinput` field to redirect the input to the pointed location when parsing the DNS name (and add an attribute `&exportsourcedata` to the `DNS_message` to make the input visible as variable `sourcedata`).

The derivative members are evaluated once during parsing and can be accessed in the same way as record or case fields in the generated C++ class. The order that derivative fields, along with non-derivative ones, are evaluated depends on only the data dependency among fields; the order is undefined for fields that do not depend on each other. (Note, this lack of ordering is deliberate, as it keeps the door open for future parallelization.) On the other hand, `binpac` provides attributes for users to introduce artificial dependency edges between fields, in case the user wants to ensure a certain ordering among evaluation of fields.

Derivative fields are also used to insert custom computation (such as event generation for the `bro` NIDS) into the parsing process, as discussed in Section 3.4.3.

3.4.2 State Management

Up to this point we have explored various issues in describing the syntax of a byte segment. To model the state of a continuous communication, `binpac` introduces notions of *flow* and *connection*. A *flow* represents a sequence of *messages* and state to maintain between messages. A *connection* represents a pair of *flows* and state between flows. Note that here *connections* are not only TCP or UDP connections, but any two-way communication sessions. For example, a DCE/RPC *connection* may correspond to a TCP connection on port 135, a UDP session to the Windows messenger port, or a CIFS "named pipe" (a DCE/RPC tunnel through the CIFS protocol).

As shown in the HTTP example (lines 38), the declaration of a connection consists of definitions of flow types for each flow. The "upflow" refers to the flow from the connection originator to the responder, and the "downflow" refers to the flow in the opposite direction. Like types, connections and flows can be parameterized, too.

Without loss of generality, we assume a flow consists of a sequence of messages of the same `binpac` type. (If a flow consists of messages of different types, we can encapsulate the types with a case type.) Thus one message type is specified for each flow, which we term *flow unit type*.

When specifying the flow unit type, we also specify how input data arrive in a flow: it may arrive as *datagrams*, each containing exactly one message, or in a *byte stream*, where the boundary of data delivery does not necessarily align with message boundaries, though the bytes are guaranteed to arrive in order.[4] The two input delivery modes are specified with keywords `datagram` and `flowunit`, respectively, as we see in the examples of DNS and HTTP parsers (lines 64 and 41 respectively).

Per-Connection/Flow State

While type parameterization allows types to share information *within* a message, in some scenarios we have to keep state at per-connection or per-flow level. For instance, a DCE/RPC parser needs to remember onto which interface a connection is bound, so that requests and replies can be parsed accordingly. As Figure 3.4 shows, a SUN/RPC parser keeps a per-connection table that maps session ID's to call parameters, and when a reply arrives, the parser can find the corresponding call parameters by looking up the reply message's session ID in the table. Connection/flow state is specified with embedded C++ code and corresponding access functions defined in `binpac` types.

Further abstraction of state is an important aspect of future work, as the abstraction can then expose data dependencies in the protocol analysis and enable better parallelization or hardware realization. The

[4]Because the flows represent abstract flows, the delivery mode of a flow does not always indicate whether the underlying transport protocol is TCP or UDP. For example, while the DNS abstract flow takes input as datagrams, it is used for both TCP and UDP, whereas in the case of TCP, an addition thin layer between the DNS and TCP protocol delimits one DNS message from another in the TCP byte stream.

```
connection RPC_Conn(bro_conn: BroConn) {
   %member{
       typedef std::map<uint32, RPC_Call *> RPC_CallTable;
       RPC_CallTable call_table;
   %}
   # Returns the call corresponding to the xid. Returns
   # NULL if not found.
   function FindCall(xid: uint32): RPC_Call
       %{
       RPC_CallTable::const_iterator it = call_table.find(xid);
       if ( it == call_table.end() )
           return 0;
       return it->second;
       %}
   function NewCall(xid: uint32, call: RPC_Call): void
       %{
       if ( call_table.find(xid) == call_table.end() )
           call_table[xid] = call;
       %}
    #...
};

type RPC_Call(msg: RPC_Message) = record {
   # ...
} &let {
   # Register the RPC call by the xid
   newcall: void = $context.connection.NewCall(msg.xid, this);
};

type RPC_Reply(msg: RPC_Message) = record {
   # ...
} &let {
   # Find the corresponding RPC call.
   call: RPC_Call = $context.connection.FindCall(msg.xid);
};
```

Figure 3.4: SUN/RPC per-connection state

main challenge in abstracting state lies in understanding which data structures, such as hash tables, FIFO queues, and stacks, are commonly used in protocol parsers and providing ways to abstract them.

The `$context` Parameter

For types to access per-connection/flow state, the references to the corresponding connection and flow have to be given to the type parse functions through function parameters. As the connection and flow might be accessed by multiple types, we can propagate them as implicit parameters to relevant types, just as the byte order flag does. More generally, state can also be maintained at granularity other than connection or flow, e.g., at a multi-connection "session" level. We aggregate all such parameters as members of an implicit *context* parameter. The members of the context parameter are declared with `analyzer <name> withcontext` at the beginning of a `binpac` specification (Figure 3.1, line 1). The member values are instantiated in the `withcontext` clause in the flow unit definition (Figure 3.1, line 42).

3.4.3 Integrating Custom Computation

In a `yacc` grammar one can embed user-defined computation, such as syntax tree generation, in the form of C/C++ code segments, which the parser executes when reducing rules. `binpac` takes a slightly different approach in integrating custom computation with parsing. The computation (e.g., generating an event in the `bro` NIDS) is embedded through adding *derivative fields* (discussed in Section 3.4.1). As the definition of type `HTTP_Header` in Figure 3.1 shows (line 94-96), a `bro` event for a HTTP header is generated by calling an external function `bro_event_http_header` in the definition of derivative field `bro_gen_hdr`. The function is invoked after parsing the data fields it depends on, `name` and `value` of the header. Note that these sorts of links are the only tie between the `binpac` specification for HTTP and the `bro` system.

3.4.4 Separation of Concerns

"Separation of concerns" is a term in software engineering that describes "the process of breaking a program into distinct features that overlap in functionality as little as possible." [127] In the case of `binpac`, one would want to separate the definition of a protocol's syntax from specifications of additional computation (such as `bro` event generation) on parsing results, because such separation allows us to reuse the protocol definitions for multiple purposes and across different systems. For the same reason, one may

```
refine typeattr HTTP_Header += &let
  process_header: bool =
      $context.flow.bro_event_http_header(name, value);
;
```

Figure 3.5: Separating `bro` event generation from protocol syntax specification with `refine`.

also want to separate specification of sub-protocols (e.g. RPC Portmapper and NFS) from the underlying protocol (e.g., RPC) and from each other.

`binpac` supports a simple but powerful syntactic primitive to allow separate expression of different concerns—parsing vs. analysis, a lower-level protocol vs. higher-level ones—and yet make the separated descriptions semantically equivalent to a unified one. The language includes a "refine typeattr" primitive for appending new type attributes, usually additional derivative fields, to existing types. For example, the generation of `http_header` event in the HTTP example (line 94-96) can be separated from the protocol syntax specification, as Figure 3.5 shows.

Such separation allows us to place related-but-distinct definitions in different `binpac` source files. A similar `refine casetype` primitive allows insertion of new case fields to a case type definition (e.g., NFS_Params as a new case for RPC_Params), facilitating syntactical separation between closely related protocols.

Note that the support for separation of concerns in `binpac` is not complete in two ways. First, one cannot easily change the set of parameters of a type (or function), which can limit extension of protocol analyzers in some cases, an area for future exploration. Second, `binpac` does not *enforce* separation of concerns, or make it easier to describe things separately than describing them together. Thus, we rely on `binpac` users practicing a discipline of separating their concerns for better code maintenance and reuse.

3.5 Parser Generation

Two main considerations in parser generation are (1) handling incremental input on many flows at the same and (2) detecting and recovering from errors. Below we examine them in turn.

3.5.1 Incremental Input

One approach to handle incremental input is to make the parsing process itself fully incremental, i.e., to make the parse function ready to stop anywhere, buffer unprocessed bytes at elementary type level, return,

```
type DCE_RPC_Header = record
  ...
  frag_length: uint16;     # length of the PDU
  ...
;

type DCE_RPC_PDU = record
  header: DCE_RPC_Header;  # A 16-byte-long header
  ...
 &length = header.frag_length;
```

Figure 3.6: Specifying buffering length of a type

and resume on next invocation. The parsing state of a composite type, such as a `record`, can be kept by an indexing variable pointing to the member to be parsed next and a buffer storing unprocessed raw data.

However, incremental parsing at elementary type granularity is expensive, because boundary checks of adjacent fields can no longer be combined. It is also unnecessary for all the protocols we have encountered. As protocols are designed for easy processing, they often have a natural unit for buffering. Binary protocols (such as DCE/RPC) often have a "length" header field that denotes the total message length. ASCII protocols are usually either line-based (such as SMTP) or alternate between length-denoted and line-based units (such as HTTP). Given such parsing boundaries, we still require support for incremental parsing, but can carry it out at larger granularity and with reduced overhead.

Thus, `binpac` provides the attributes `&length` and `&oneline` to specify buffering units[5]. `&length` gives a message's length in bytes while `&oneline` triggers line-based buffering. `&length` usually points to a corresponding length field in the header (Figure 3.6) but can generally take any expression to compute the length. The `binpac` compiler performs data dependency analysis to find out the initial number of bytes to buffer before the length expression can be computed (in the case of a DCE/RPC message, the first 16 bytes). The generated code will buffer the message in two steps, first the initial bytes for computing the message length, then buffer up to the full length before parsing the remaining fields.

3.5.2 Error Detection and Recovery

Protocol parsers have to robustly detect and recover from various kinds of errors. Errors can be caused by irregularity in real-world traffic data, including small syntax deviations from the standard, incorrect length fields, corrupted contents, and even payloads of a completely different protocol running on the standard

[5]`binpac`'s incremental analysis depends on the existence of these attributes. Viewing the record definitions as a tree of types, each path from the root type to a leaf must contain one of them at a non-leaf node.

port of the parsed protocol. Errors can also result from incomplete input, such as due to packet drops when capturing network traffic. In these cases, the parser might not know in the specific state of the dialog, e.g., whether what it now sees on HTTP flow is inside a data transfer or not. Errors may also arise through incorrect `binpac` specifications, e.g., through missing cases or trying to access an unparsed case field, or due to adversarial manipulation, as discussed earlier.

Parsers generated by the `binpac` compiler detect errors of various aspects, as we discuss below. When an error is detected, the code throws a C++ run-time exception, which can then be caught for recovery.

Error Detection

Efficient Boundary Checking. Conceptually, boundary checking (whether scanning stays within the input buffer) only need take place before evaluating every elementary integer or character type field, because all other types are composed of elementary types. While it would be easy to generate the boundary checking code this way, the generated code would be quite inefficient. Instead, the `binpac` compiler tries to minimize the number of boundary checks. The basic idea is: before generating boundary checking code for a record field, check recursively whether we can generate the checking for the next field. If so, we can combine them into one check. In this way, the compiler can determine the furthest field for which the boundary checking can be performed at a given point of parsing.

Handling dropped packets. When capturing network traffic, packet drops cannot always be avoided. They can be caused by a high traffic volume, kernel scheduling issues, or artifacts of the monitoring environment. Such drops lead to *content gaps* in application-level data processed by protocol parsers. Facing content gaps, parsers not only are unable to extract data for the current message, but also may not even know where the next message starts.

A particular, very common case of a content gap is one located inside a byte sequence of known length. For example, within an HTTP entity body, a content gap can be handled without creating uncertainty for subsequent protocol elements. If a byte sequence is defined as `&chunked` in a `binpac` specification—and thus only passed to a potential `&processchunk` function, but not further referenced by other expressions—then the generated parser can simply skip over such a gap. (If `&processchunk` is defined for the sequence, the function is called with a specially marked “gap chunk” so it can take note of the fact.) This mechanism allows us to handle most content gaps for protocols in which the majority of data is contained in long byte sequences. Hand-written protocol parsers in `bro` handle content gaps in a similar way, but on an individual basis; the chunked byte string abstraction in `binpac` allows them to be handled universally for all protocols.

In general, it is trickier to handle content gaps which do not fully fall into a byte sequence of known size. We discuss these below in Section 3.5.2.

Run-time type safety. The only access to parsing results provided to `binpac` parsers is via typed interfaces. These leaves two aspects of type safety to enforce at run-time: (1) among multiple case fields in a case type, the generated code ensures that only the case that is selected during parsing can be accessed, otherwise it throws a run-time exception; (2) access to array elements is always boundary-checked. On the other hand, note that `binpac` cannot guarantee complete safety, as it allows arbitrary embedded C++ code which it cannot control.

User-defined error detection A user may also define protocol-specification error checking, using the `&check` attribute. For example, one may check the data against some protocol signature (e.g., the first 4 bytes of a CIFS/SMB message should be "`\xffSMB`") to make sure the traffic data indeed reflects the protocol.

Error Recovery

Currently errors are handled in a simple model: when the flow processing function catches an exception, it logs the error, discards the unfinished message as well as the unprocessed data, and initializes to resume on the next chunk of data.

One potential problem with this approach is that, for stream-based protocols, the next message might not be aligned with the next payload chunk. In the future we plan to add support for re-discovering message boundaries in such cases. Having such a mechanism will also help to further improve parsing performance, as we can then skip large, semantically uninteresting messages, and re-align with the input stream afterwards.

Protocol	Hand-written			binpac		
	LOC	Time (seconds)	Throughput	LOC	Time (seconds)	Throughput
HTTP	1,896	538–541	244 Mbps / 36.7 Kpps	676	442–444	298 Mbps / 44.7 Kpps
DNS	1,425	37.3–37.5	18.6 Mbps / 13.3 Kpps	698	44.7–44.8	15.6 Mbps / 11.1 Kpps

Table 3.2: Number of lines of code (LOC), CPU time, and throughput of protocol analyzers

3.6 Experiences with `binpac`

We have used `binpac` to add protocol parsers for CIFS/SMB, DCE/RPC (including its end-point mapping protocol) and NCP to `bro`'s traffic analysis engine.[6] To compare `binpac` with handwritten protocol parsers, we also rewrote the parsers for HTTP and DNS (and SUN/RPC, which we have not yet evaluated) in `binpac`. We use these latter to provide a comparison in terms of code size and performance between `binpac`-based and hand-written parsers.

As Table 3.2 shows, the `binpac`-based parsers for HTTP and DNS have code sizes of roughly 35–50% that of the hand-written parsers, measured in lines of code (and the same holds in source file sizes), respectively. We also note that for both protocols, the Bro-specific semantic analysis comprises well over half of the `binpac` specification, so for purposes of reuse, the specifications are significantly smaller than shown.

To test the performance of the parsers, we collected a one-hour trace of HTTP and DNS traffic at Lawrence Berkeley National Laboratory's network gateway. The HTTP subset of the trace spans 19.8M packets and 16.5 GB of data. The DNS subset spans 498K packets and 87 MB. The drop rate reported by tcpdump when recording the trace was below 4×10^{-6}.

Table 3.2 shows the CPU time required for each type of analysis, giving the minimum and maximum times measured across 5 runs, using a 3.4 GHz Xeon system running FreeBSD 4.10 with 2 GB of system RAM. We also show the throughput in bits/sec and packets/sec, observing that on a per-packet basis, DNS analysis is much more expensive than HTTP analysis, since many HTTP packets are simply entity data transfers requiring little work.

For these numbers, we disabled Bro's script-level analysis of the protocols, so the timings reflect the computation necessary for the parser to generate the Bro events corresponding to the application activity (including TCP/IP processing and TCP flow reassembly) with no further processing of those events.

[6] Given the complexity of CIFS, the parser does not yet cover the entire protocol, but only the commonly seen message types.

We see that the binpac HTTP parser performs significantly better than the hand-written one. This gain came after tuning the specification by adding a &transient attribute to HTTP_header fields, which instructs binpac to not create a copy of the corresponding bytestring, instead, bytestrings will point directly to portions of the input buffer. (Therefore transient strings are visible only within the parsing function of the type, while non-transient ones, which are copied, can be accessed after parsing.) We have not yet applied the same tuning to the DNS specification; as a result, it allocates many more dynamic objects, and copies more strings than the hand-written one does. We do, however, believe that tuning it will prove straightforward.

We also note that in developing our DNS parser we found two significant bugs in the hand-written parser's processing. These related to using incorrect field widths and non-portable byte-ordering manipulations, and provide direct examples of the benefit in terms of correctness for specifying analyzers in a high-level, declarative fashion.

3.7 Summary and Future Directions

This chapter presents binpac, a declarative language for generating parsers of application-layer network protocols from high-level specifications. Such parsers are a crucial component of many network analysis tools, yet coding them manually is a tedious, time-consuming, and error-prone task, as demonstrated by the numerous severe vulnerabilities found in such programs in the past.

binpac reflects a different paradigm for building protocol parsers: abstracting their syntax into a high-level meta-grammar, along with associated semantics. A parser generator then translates the specification into low-level code automatically. By providing such an abstraction, a programmer can concentrate on high-level protocol aspects, while at the same time achieve correctness, robustness, efficiency and reusability of the code.

In spirit, this approach is similar to that embodied in the use of yacc for writing parsers for programming languages, but many elements of the network analysis problem domain require significantly different underlying mechanisms. First, there are critical differences between the syntax and grammar of network protocols and context-free languages. In addition, processing network traffic requires a fundamentally different approach in terms of handling input, namely the ability to incrementally parse many concurrent input streams.

Our domain-specific `binpac` language addresses these issues with a set of network-specific features: parameterized types, variable byte ordering, automatic generation of boundary checking, and a hybrid approach of buffering and incremental parsing for handling concurrent input. `binpac` supports both binary and ASCII protocols, and we have already used it to build parsers for HTTP, DNS, SUN/RPC, RPC portmapper, CIFS, DCE/RPC (including the endpoint mapper), and NCP. We integrated all of these into the Bro NIDS, replacing some of its already existing, manually written ones. Our evaluation shows that `binpac` specifications are 35–50% the size of handcoded ones, with the protocol description (independent of the user's analysis semantics) comprising less than half of the specification. Our HTTP parser runs faster than the handcrafted one it replaces (and with equal memory consumption), and we are confident that the DNS will likewise soon exhibit performance equal to the one it replaces. `binpac` is open-source and now ships as part of the Bro distribution.

In the future, along with specifying further protocols in `binpac`, we envision exploiting its power in two areas. First, we wish to explore the reusability of `binpac`-generated parsers by integrating them into additional network tools. Second, we intend to add back-ends other than C++ to `binpac` to generate parsers for different execution models. As proposed in [85], we specifically aim to build highly parallel parsers in custom hardware.

Chapter 4

A Programming Environment for Trace Anonymization

Packet traces of operational Internet traffic are invaluable to network research, but public sharing of such traces is severely limited by the need to first remove all sensitive information. Current trace anonymization technology leaves only the packet headers intact, completely stripping the contents; to our knowledge, there are no publicly available traces of any significant size that contain packet payloads. This chapter describe a new approach to transform and anonymize packet traces. Our tool provides high-level language support for packet transformation, allowing the user to write short policy scripts to express sophisticated trace transformations. The resulting scripts can anonymize both packet headers and payloads, and can perform application-level transformations such as editing HTTP or SMTP headers, replacing the content of Web items with MD5 hashes, or altering filenames or reply codes that match given patterns. The chapter also discusses the critical issue of verifying that anonymizations are both correctly applied and correctly specified, and experiences with anonymizing FTP traces from the Lawrence Berkeley National Laboratory for public release.

The rest of this chapter is organized as follows. The next section presents background and problem statement. Section 4.2 enumerate goals of this work. Section 4.3 describes generic packet trace transformation. Section 4.4 explores issues in trace anonymization. Related work is discussed in Section 4.6. The chapter concludes with a summary in Section 4.7.

4.1 Problem Statement

Researchers often use tools such as `tcpdump` to capture network packet traces. Packet traces recording real-world Internet traffic are especially useful for research on traffic dynamics, protocol analysis, workload characterization, and network intrusion detection. However, sharing of Internet packet traces is very limited because real-world traces contain many kinds of sensitive information, such as host addresses, emails, personal web-pages, and even authentication keys. The traces must be first "anonymized" to eliminate any private information, for example, IP addresses, user IDs, and passwords, before they can be shared among researchers.

To date, Internet packet trace anonymization has been limited to only retaining TCP/IP headers [132, 91], with IP addresses renumbered and packet payloads completely removed. The lack of traces with application layer data greatly limits research on application protocols. It is especially crippling for network intrusion detection research, forcing researchers to devise synthetic attack traces that often lack the verisimilitude of actual traffic in critical ways, resulting in errors such as grossly underestimating the false positive rate of "anomaly detection" techniques. [61, 9]

In this work we develop a new method to allow anonymization of packet payloads as well as headers. Traces are processed in three steps:

1. Payloads are reassembled and parsed to generate application-protocol-level, semantically meaningful data elements.

2. A policy script transforms data elements to remove sensitive information and sends the resulting elements to the composer.

3. The trace composer converts application protocol data elements back to byte sequences and frames the bytes into packets, matching the new packets to the originals as much as possible, in order to preserve the transport protocol dynamics.

Parsing allows the trace transformation policy script to operate on semantically meaningful data elements, such as user names, passwords, or filenames, making policy scripts more concise and comprehensible than those operating directly on packets or byte sequences. Working at a semantic level also gives the opportunity for less draconian anonymization policies. For example, the added information that the string "`root`" appears in a filename ("`/root/.cshrc`") rather than as a user name might, depending on a site's anonymization policy, allow the string to appear in an anonymized trace, whereas a purely textual anonymization would have to excise it, because it could not safely verify that the occurrence did not reflect a user name.

The design of trace composer aims to generate "correct" traces, for instance, as payload data is modified, checksums, sequence numbers, and acknowledgments are adjusted accordingly. The output traces just look as if they were collected from the real Internet, except that they do not carry private information. Accordingly, analysis tools that work on raw traces will likewise work on the anonymized traces.

In order to make the anonymization process amenable to validation, we follow a "filter-in" principle throughout our design of the anonymizer: instead of focusing on "filtering out" sensitive information, the anonymizer focuses on what, explicitly, to *retain* (or insert, in a modified form) in the output trace. With this principle, it becomes much easier to examine a policy script for privacy holes.

An optional "manual inspection" phase can keep more non-sensitive information in the output trace as the general anonymization script may have to make conservative judgments for some data elements; for example, whether to allow the command "`UUSER`" to appear in a trace of anonymous FTP traffic (the presence of such a typo can be useful for some forms of analysis, such as anomaly detection).

We implemented the anonymizer as an extension to `bro` [89], a network intrusion detection system, to take advantage of its application parsers and its built-in language support for policy scripts.

Beside anonymization, our tool can also be used for generic trace transformations, providing a great degree of freedom and convenience for various types transformation. For example, we can take a trace of FTP traffic and remove from it all the connections for which the user name was not "`anonymous`"; or all the ones for which the FTP authentication was unsuccessful; or those that do uploads but not downloads. A different type of transformation is for testing network intrusion detection systems by inserting attacks into actual background traffic by slightly altering existing, benign connections present in a trace. Still another type of transformation is to remove large Web items from HTTP connections (including persistent sessions with multiple items) in order to save disk space (see Section 4.3.4).

In a sense, the tool spells the end of traces as being stand-alone evidence of any sort of application-level network activity, since it makes it so easy to modify what a trace purports to show.

We developed trace transformations for FTP, SMTP, HTTP, Finger, and Ident. As a test of the approach, we anonymized FTP traces from the Lawrence Berkeley National Laboratory (LBNL). Besides testing the technology, one of the important questions behind the exercise was to explore what sort of anonymizations a site might require, and being willing to abide, for public release of traces with contents. To this end, working with the site we devised an anonymization policy acceptable to the site and approved for public release. The corresponding traces are available from [79].

4.2 Goals

We designed the transformation tool with the following goals in mind:

1. Policy scripts operate on application-protocol-level data values. This means that instead of operating on packets or TCP flows, a policy script sees typed and semantically meaningful values (e.g., HTTP method, URI, and version). Likewise, the trace transformation scripts also specify application-protocol-level data to the output trace, without needing to dictate the details of generating the actual packets.

2. The output traces contain well-formed connections: packets have correct checksums and lengths, TCP flows can be reassembled from the resulting packets, and application-protocol data has correct syntax[1], so that other programs can process the transformed traces in the same way that they handle original `tcpdump` traces.

3. The mechanism supports generic trace transformations besides anonymization.

4. The anonymization is "fail safe" and amenable to verification. Fail-safety means that the privacy resulting from the anonymization does not depend on the tool and the policy script being completely correct. Being amenable to verification means it is easy to examine and validate the policy script, the anonymization process, and the output trace.

The first and third goals dictate where to separate mechanism and policy: (1) the mechanism part should parse the input trace to *expose* all application-protocol semantic elements, e.g., commands, reply codes, MIME header types; (2) the mechanism should not restrict how the values are changed, but leave that to the policy script. We discuss mechanism and anonymization policy in the next two sections, respectively.

[1]Or not, if the policy script decides to keep the "dirtiness" of the original trace.

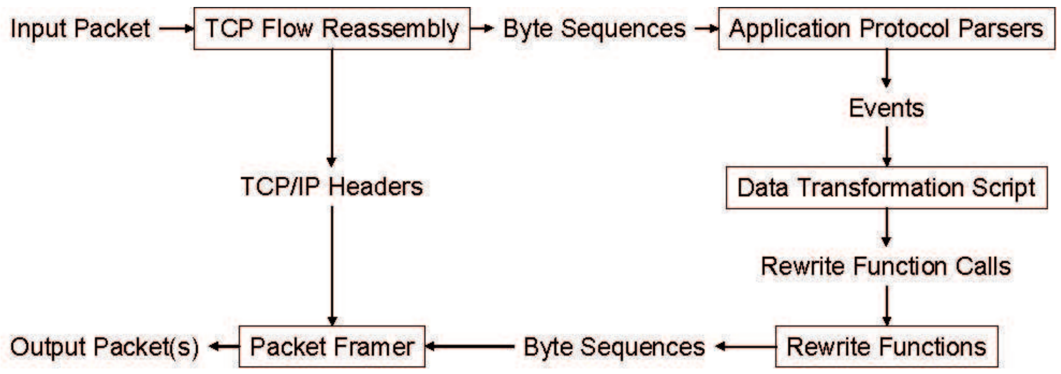


Figure 4.1: Data Flow in Trace Transformation

4.3 Generic Trace Transformation

Trace transformation consists of three steps: parsing, data transformation, and composition. These are shown as the right-hand components of Figure 4.1. The parsing and composition parts do not depend on the type of trace transformation, and we have implemented them in `bro` as built-in mechanisms. The second step (data transformation) is fully programmable, however, and so is implemented as a `bro` policy script.

We first look at the process from the viewpoint of the policy script, focusing on the trace input/output interface, and then discuss details of trace parsing and composition.

4.3.1 Policy Script Programming Environment

The `bro` policy script language is procedural, with strong typing that includes support for several network-specific types (e.g., addresses and ports), as well as relative and absolute time, aggregate types (hash tables and records), regular expression matching, and string manipulation. More details about the `bro` language can be found in [89, 86].

From the point of view of a policy script, the parsing part is `bro`'s event engine, and the composer is a family of library functions, which we call "rewrite functions".

A policy script for a protocol usually contains several "event handlers", which are execution entry points of the script. Through event parameters, each event handler receives protocol-semantic data elements as well as a record corresponding to the particular TCP connection. An event handler may call other functions to process the data, and writes the transformed data to the output trace by calling the rewrite functions.

When calling a rewrite function, the policy script specifies a connection, and sometimes also direction of the flow, to write the data to. The destination connection is usually the same connection of the event, but can also be any other connection present in the input trace at the same time.

For example, a line in an SMTP message "`MAIL From:<alice@bob.org>\r\n`" arriving on connection C generates the following event:

```
smtp_request(
    conn: connection = C,
    command: string = "MAIL",
    argument: string = "From: <alice@bob.org>")
```

The policy script receives the command and argument and decides what to write to the output trace—e.g., it could call:

```
rewrite_smtp_request(C, "MAIL", "From: <name123@domain111>")
```

to change the sender in the trace from "`alice@bob.org`" to "`name123@domain111`".

There is usually a correspondence between protocol events and rewrite functions: e.g., for event `smtp_request`, there is function `rewrite_smtp_request`, and they have the same or very similar set of parameters.

Explicit Rewriting. Note that the trace composer API requires explicit rewrites, i.e., for a data element to get into the output trace, it must be explicitly placed there by the policy script calling a rewrite function. Alternatively, another style we could have chosen for the composer API would be to have the policy script only specify data elements that should be *changed*, and pass the rest through unmodified. With this style, we could implement a single generic interface by which scripts would directly specify the element to change. For example, the SMTP rewrite above would be specified as:

```
modify_element(smtp_request_arg, "From: <name123@domain111>")
```

and the composer would alter the location in output trace occupied by the variable `smtp_request_arg` to contain the new text rather than the original.

While appealing because a single rewrite function would suffice for all protocols (though the application parsers would have to annotate each script variable with its location in the connection's byte stream), instead of having a family of rewrite functions for various protocols, we choose the heavier API because it presents a safer interface for trace anonymization. First, requiring explicit rewrite forces the policy script writer to put consideration into every element, so it will be less likely that they overlook a privacy hole. Second, it is easier for other people to examine a policy script for privacy leaks, as the examiner only needs to look at elements written in the script (rather than having to keep in mind all the protocol elements that are implicitly not being changed because they don't show up in the script). This design choice shows how the "filter-in" principle affects our design. Additionally, this interface allows type-checking on trace-rewrite operations to catch inconsistency between output data elements.

4.3.2 Trace Parsing

Trace parsing usually consists of three steps: flow reassembling, (optional) line breaking, and protocol-specific parsing.

Flow Reconstruction. `bro`'s application parsing begins by reassembling IP fragments and then reassembling the TCP byte stream. (We ignore here `bro`'s UDP processing, though our techniques could be applied to it, too.) In case of TCP retransmission or packet reordering, the bytes that arrive first are not delivered until the gap is filled, at which point the bytes are delivered together. For example, suppose an SMTP command arrives in three packets with the last two in reverse order: "`MAIL Fro`", "`bob.org>\r\n`", and "`m:<alice@`". The reassembler will emit "`MAIL Fro`" on the first packet arrival, nothing on the second because it comes out of order, and "`m:<alice@bob.org>\r\n`" after processing the third packet.

Breaking into Lines. Many protocols (e.g., SMTP, FTP, the non-data part of HTTP) process application data one line at a time. For such protocols, there is an intermediate step that structures the bytes from reassembler into lines before protocol-specific parsing. Following the above example, the line divider will emit a line "`MAIL From:<alice@bob.org>`" after it sees `\r\n`.

Protocol-Specific Parsing. The parser takes plain bytes as input and emits typed and semantically meaningful data fields. It first divides the bytes according to protocol syntax, then converts bytes of each field to typed values—e.g., string, integer, boolean, record—and groups the values by events, finally placing the events in an event queue. (As event parameters, each data element carries a semantic meaning.) Currently `bro` has parsers for most commonly seen protocols, including DCE/RPC, DNS, FTP, HTTP, MIME, Netbios, NFS, Rlogin, RPC, SMB, SMTP, SSH, and Telnet.

A major challenge in parsing is that the parser often cannot strictly follow the RFCs that define the application protocol, since in practice there are frequently deviations from the letter of the standards, or deficiencies in the traffic being analyzed. Two particular difficulties relevant for our discussion are:

Line Delimiters Line-oriented protocols (e.g., SMTP, HTTP) generally are specified to use the two-byte sequence CRLF (`\r\n`) as the delimiter between lines. However, some end hosts also interpret single LF (`\n`) and/or CR (`\r`) as the end of the line. Ideally, we would like to identify which delimiter each host uses, and consistently apply that interpretation.

Content Gaps. For traces captured under high-volume traffic conditions, sometimes the packet filter fails to capture all of the packets. Such "content gaps" are generally unsolvable, but we found that most of them occur within the data-transfer section of an application dialog rather than in the command/reply exchange. We developed a content gap recovery mechanism for SMTP and HTTP that skips over gaps that appear consistent with being wholly contained within a data transfer. With this heuristic, we find that most content gaps no longer disrupt parsing. (We note that content gaps are also delivered as events, and the policy script may decide to eliminate them or keep the gaps in the output trace.)

In summary, there can be some loss of fidelity when data goes through the trace parser. This is in fact a general problem for any network monitoring tools.

4.3.3 Trace Composer

The trace composer consists of rewrite functions and a packet generator. As discussed above, the rewrite functions are called during event processing. A rewrite function generates a byte string on each invocation and buffers the string for the packet generator. `bro` then invokes the packet generator to process buffered bytes and generate output packets. Below we look at the rewrite functions and packet generation in detail.

Rewrite Functions

A rewrite function performs the inverse of parsing: it prints the typed data elements to a byte string in a protocol specific format, placing them in the right order and adding proper delimiters. For example, `rewrite_finger_request` takes four parameters: `c` (the associated connection, of type `connection`, which is a record of connection information), `full` (a boolean flag indicating whether the Finger request was for the "full" format), `username` and `hostpart` (both strings). The rewrite function concatenates `username` and `hostpart`, adds `\r\n` to the end, and inserts "`/W `" to the beginning of the line when `full` is true. Thus, with parameters `(T, "alice", "host123")`, the function generates the string "`/W alice@host123\r\n`", and with parameters `(F, "bob", "")`, it generates "`bob\r\n`".

Rewrite Function Compiler. When implementing the rewrite functions for various protocols, we found a number of commonalities: they all need to convert `bro` values to C++ native values and fetch the connection object, and for each built-in function we need to write a `bro`-language prototype declaration and add initialization code to bind the `bro` built-in function to the C++ function. So we looked for ways to facilitate code reuse to avoid the tedious and error-prone task of repeating the similar code at each place.

To do so, we developed a "rewrite function compiler". We write rewrite functions with `bro`-style function prototypes and C++ bodies. The compiler inserts code for the value conversion and connection record fetch, extracts `bro` function prototypes, and generates function binding code. With the rewriter compiler, most rewrite functions can be implemented with around 10 lines of code each. Figure 4.2 shows the source code and the resulting C++ code of "`rewrite_finger_request`". Note that each rewrite function has a hidden first parameter: "`c: connection`", which is inserted into the C++ code and the `bro` prototype during compilation.[2]

Currently we have implemented rewrite functions for FTP, HTTP, SMTP, Finger, and Ident.

Packet Generation: Framing

After rewriter functions emit byte sequences, the *packet framer* decides how to pack the bytes into packets. It cares about (1) whether the bytes should fit into a single packet or be split across multiple ones, and (2) what timestamp to attach to each packet.

The central concern of the packet framing algorithm is to keep the traffic dynamics as close to the original as possible and yet to remain transparent to the policy scripts. For example, an HTTP request can be transmitted line-by-line, one packet per line, or all in one packet; for each of these cases, we would like the rewritten request to maintain the original packet structure and the timestamps.

Note that we cannot directly reuse the packet structure—sizes and ordering—of the input trace because there is not necessarily a one-to-one mapping between bytes in the input and output traces, as a policy script can change data lengths, insert or remove objects, or change the ordering among objects. So in general it is only possible to *approximate* the original dynamics. Also, because the policy script does not

[2]The boolean variable "is_orig = 1" means the direction of the TCP flow is from the connection originator (the Finger client).

[Source Code]

```
# Write a finger request to trace.
rewriter finger_request %(full: bool,
                username: string, hostpart: string%)
        %{
        const int is_orig = 1;
        if ( full )
                @WRITE@(is_orig, "/W ");
        @WRITE@(is_orig, username);
        if ( hostpart->Len() > 0 )
                {
                @WRITE@(is_orig, "@");
                @WRITE@(is_orig, hostpart);
                }
        @WRITE@(is_orig, "\r\n");
        %}
```

[Resulting C++ Code]

```
Val* bro_rewrite_finger_request(val_list* BiF_ARGS)
{
    if ( BiF_ARGS->length() != 4 )
        {
        run_time("finger_request() takes exactly 4 argument(s)");
        return 0;
        }
    TCP_Rewriter* trace_rewriter = get_trace_rewriter((*BiF_ARGS)[0]);
    if ( ! trace_rewriter )
        return 0;
    int full = (int) ((*BiF_ARGS)[1]->AsBool());
    StringVal* username = (StringVal*) ((*BiF_ARGS)[2]->AsStringVal());
    StringVal* hostpart = (StringVal*) ((*BiF_ARGS)[3]->AsStringVal());

    const int is_orig = 1;
    if ( full )
        trace_rewriter->WriteData(is_orig, "/W ");
    trace_rewriter->WriteData(is_orig, username);
    if ( hostpart->Len() > 0 )
        {
        trace_rewriter->WriteData(is_orig, "@");
        trace_rewriter->WriteData(is_orig, hostpart);
        }
    trace_rewriter->WriteData(is_orig, "\r\n");

    return 0;
} // end of finger_request
```

Figure 4.2: Source and the Resulting C++ Code of a Rewrite Function

explicitly specify the mapping between original and new data objects, when it calls rewrite functions, the trace composer has to derive an implicit temporal mapping from bytes to packets, as follows.

In the common case, transformed data is written to the same TCP flow (i.e., same direction of a TCP connection) as the input packet currently being processed. The framer places the bytes in the *current output packet*. If the payload size exceeds the MTU, it generates another output packet with the same timestamp to hold the rest of the data.

Usually the data written by the policy script originates from data in the current input packet; thus, the output trace has a similar packet structure as the input trace. However, there are two cases in which the data to write actually comes from an earlier or later input packet:

1. When an event consists of data from multiple packets, the data may range across packet boundaries or appear in retransmitted packets. In this case, the transformed data will be written with respect to the last packet associated with the event, i.e., the packet whose arrival makes the trace parser generate the event.

2. In some cases, the policy script cannot decide immediately what to write before seeing later data. For example, when rewriting HTTP messages, the new Content-Length header for an HTTP entity cannot be decided until the entity is entirely transformed.

For the first of these, we find it tolerable to simply associate the data with the event's last packet, because to do otherwise would require a great deal of work—tracing each event parameter's origin throughout the trace reassembly and parsing hierarchy in order to know from exactly which input packet the data originates.

Deferring Writes. The second case, of the policy script having to defer its transformation decision, presents a larger problem, because it not only leads to imprecise timestamps for output packets, but also causes inconvenience for transformation script programming: in the HTTP message case, the Content-Length header has to be written before the data entity, so the script must buffer up all the transformed data entity until it finishes processing the entire entity. To address this problem, we added support for deferring writes so that the script can essentially write packets out of order.

The trace composer supports deferring writes by allowing the policy script to reserve slots in current output packets. The script may then seek the reserved slot at a later point, write data to it, and release the slot. (Figure 4.3)

```
# Reserve a slot when the original Content-Length header
# arrives on connection c
msg$header_slot = reserve_rewrite_slot(c);

...

# After the entire data entity is processed, seek the
# slot
seek_rewrite_slot(c, msg$header_slot);

# Write the header to the slot
rewrite_http_header(c, is_orig, "Content-Length",
                      fmt(" %d", data_length));

# And release the slot
release_rewrite_slot(c, msg$header_slot);
```

Figure 4.3: Deferring Writes to HTTP Content-Length Header

Packet Generation: TCP/IP header fields

Once packet payloads are determined, the trace composer attaches TCP and IP headers to output packets. Also, if no data is written in the current packet cycle, but the trace composer needs to construct a packet to carry a TCP flag (SYN, RST, or FIN) or simply an acknowledgment, it generates an empty packet and attaches the headers to the packet.

For every output packet, the trace composer first fetches the TCP and IP headers of the most recent input packet on the same TCP flow and generates the new headers by modifying the following header fields:

1. If the trace is being anonymized, the source and destination addresses in the IP header are anonymized, as discussed in Section 4.5.2.

2. As the output trace does not have IP fragments (`bro` reassembles fragments early in its protocol processing, making it too difficult to track their contribution to the final byte stream), the composer clears fragment bits in the IP header.

3. The composer keeps the original IP identification field, unless the (source IP, ID) pair has already appeared in the output trace, in which case we increment the ID till no conflict is found.

4. TCP sequence/acknowledgment numbers are adjusted to reflect new data lengths, as is the IP packet length field. The composer then recomputes the TCP and IP checksums. (Note that, similar to the

case of fragments in the input trace, because `bro` discards packets with checksum failures early in its processing, it is too difficult to propagate checksum errors into the transformed output.)

5. Currently the composer discards IP options, because `bro` lacks an interface to access them, and some of them would take significant effort to address. The composer keeps certain TCP options, such as maximum segment size, window scaling and SACK negotiation (but not SACK blocks, due to the ambiguity of the location of the SACK'd data in the transformed stream), and timestamps; and replaces other options with the NOP option.

6. TCP flags are propagated, except that the composer removes the FIN flag. This is because additional packets may be inserted after the last one present in the input stream, and these must still be numbered in the sequence space before the final FIN to comply with TCP semantics. We can imagine a "conceptual" FIN that is reordered together with the payloads and comes only at the end of the data flow. Therefore, the trace composer inserts a FIN flag only when the flow reassembler has delivered, and the transformation script has processed, the last chunk of the flow.

4.3.4 Trace Rewriters for Trace Size Reduction

As a demonstration of the utility of trace transformation in addition to anonymization, we implemented trace rewriters for HTTP and SMTP to reduce the *size* of traces rather than the privacy of their embedded contents. At LBNL, for example, the volume of HTTP traffic often exceeds 50 GB per day. The site wants to continuously record this traffic (for intrusion detection analysis), but the volume proves problematic.

HTTP trace rewriter replaces HTTP entities beyond a specified size with their MD5 hash values, changes the Content-Length header to reflect the new data length, and keeps the original Content-Length and the actual data length in an "X-Actual-Data-Length" header (see Appendix 7.3 for an example). Testing it on a 729 MB trace file, and setting the threshold to 0 bytes (so all entities are replaced by hashes), the rewriter reduces the trace size to 25 MB, a factor of 29. If we compare the *gzipped* sizes of the traces (which the site often does with traces, in order to keep them longer before the disk fills up), the reduction becomes a factor of 69 (from 377 MB to 5.5 MB). Alternatively, we can implement more selective size reductions, such as stripping out only non-HTML objects in order to keep the cross-reference structure intact. Operationally, the site keeps the first 512 bytes of each entity, and keeps those with a MIME type of "text" in their entirety; this results in about a factor of 10 in size savings, yet retains enough information for intrusion analysis—one can analyze most HTTP attacks in the resulting traces to determine whether the attacks succeeded.

SMTP trace rewriter replaces mail bodies with MD5 hash values and size information, but keeps all SMTP commands/replies and mail headers.

4.4 Trace Anonymization

In this section we discuss general issues in trace anonymization and analyze four types of possible attacks against anonymization. against any anonymization scheme is inevitably dependent on the specific policy approved by the site, the general techniques are often applicable to many sites and protocols.

4.4.1 Objectives of Anonymization

The information we try to hide through anonymization falls into two categories: *identities*, including identity of users, hosts, and data; and confidential *attributes*, e.g., passwords, or specifics of sensitive user activity [30].

The first step of developing an anonymization scheme is to decide what information in the trace we need to hide. For example, in anonymizing FTP traces, we aim to hide *identities* of clients, private data (hidden files), and private servers; and sensitive *attributes*: e.g., passwords, authentication keys, and in some cases filenames.

Confidential information can be exposed via direct means, or *inferred* via indirect means. Therefore, to hide the identity of client hosts, it may not be enough to just anonymize their IP addresses. We analyze four kinds of inference attacks that may reveal confidential information through indirect means, but before doing so we first discuss the anonymization primitives, i.e., how we anonymize basic data elements.

4.4.2 Anonymization Primitives

Constant Substitution. One way to anonymize a data element is to substitute the data with a constant, e.g., replace any password with the string “`<password>`”.

Constant substitution is usually used to anonymize confidential attributes. Applying constant substitution to identifiers (e.g., IP addresses), however, is generally undesirable, as we would then no longer be able to precisely distinguish objects from one another. Instead, identifiers are usually anonymized with a 1-1 mapping, such as sequential numbering or hashing, so that the anonymized identifiers are still unique, as follows.

Sequential Numbering. We can sequentially number all *distinct* identifiers in the order of appearance, e.g., mapping files names to "`file1`", "`file2`", etc.

Hashing. One shortcoming of *sequential numbering* is that we have to keep the whole mapping history to maintain a consistent mapping during the anonymization process and across anonymizations. Instead, we can use a hash function as the mapping. Doing so requires no additional state during the anonymization process, and in addition using the same hash function across anonymizations will render a consistent mapping (assuming that the range of the hashing function is large enough so that likelihood of collision is negligible). To preserve confidentiality, the hash function must be one-way and preferably resistant to chosen plain-text attack, so that an adversary can neither discover the input from the output nor compute the hash by themselves. HMAC-MD5 (with a secret key) satisfies these requirements. Assuming the adversary can neither reverse MD5 nor extract the secret HMAC key, *hashing* is as secure as *sequential numbering*.

Prefix-Preserving Mapping. Sometimes it is valuable to preserve some of the structural relationships between the identifiers, which *sequential numbering* and *hashing* cannot do. For example, IP addresses can be anonymized in a prefix-preserving way [66, 132] such that any two IP addresses sharing a prefix will share a prefix of the same length in their anonymized form. Prefix-preserving mapping can be similarly applied on the directory components of file names. While being valuable for some forms of analysis, prefix-preserving mapping also reveals more information about the identifiers and thus is more vulnerable to attacks [135].

Adding Random Noise. We can add noise to numeric values, e.g., file sizes, to make the result more resistant to fingerprinting attacks such as matching file sizes in the trace with public files [114]. We have not applied this primitive in our experiments, however, so we do not have experience regarding how effective it is and the degree to which it diminishes the value of the trace.

4.4.3 Inference Attacks

Besides anonymizing certain identifiers and attributes to eliminate direct exposure of identities and secret data, we also consider rewriting other data fields to prevent *indirect exposure*. In order to understand which data should be anonymized, we need to analyze how an adversary might use additional data to infer confidential information. Below we discuss four kinds of inference techniques and how they relate to our FTP anonymization efforts.

Fingerprinting

"Fingerprinting" is the notion of an adversary recovering the identity of an object by comparing its attributes to attributes of objects known by the adversary. In order to do so the adversary has to know the fingerprints of the candidate objects. Thus, they cannot, for example, discover a previously unknown FTP server through fingerprinting.

We present here a brief analysis of possible fingerprinting on our anonymized FTP traces, to convey the flavor of problem:

1. Fingerprinting files: possible for public files, by looking for matches in file sizes, similar in spirit to the techniques of Sun et al [114].

2. Fingerprinting servers: possible for public servers, by the structure of their reply messages (especially the `220` greeting banner), help replies, SITE commands, or through fingerprinting files on the server. It is unclear to us whether it is possible to fingerprint servers by analyzing response timing.

3. Fingerprinting clients: there are at least two possible ways to fingerprint clients: (1) when the client displays some peculiar behavior known to the adversary; (2) through "active" fingerprinting: the adversary inserts a fingerprint for a certain client by sending packets to the trace collection site with a forged source address of the client's host address, and then looks for how these were transformed in the anonymized trace.

While fingerprinting public files and servers can expose usage patterns, this does not appear to be a serious issue because *who* made the access is not exposed.

Fingerprinting clients, on the other hand, would in some circumstances pose a significant privacy threat. But this is generally difficult for the adversary to accomplish. For the first type of fingerprinting, the client's sessions must possess peculiarities that survive the anonymization process, and the adversary must discover these. For the second type of fingerprinting, the fingerprint has to be inserted during trace collection. We discuss a defense against active fingerprinting, “knowledge separation”, in Section 4.4.3.

A particular threat is that a class of clients displaying certain peculiar behaviors will stand out from other clients. If we want to eliminate this threat, we should eliminate or blur the distinction among client behaviors—which might significantly reduce the value of the trace.

Structure Recognition

Similar to fingerprinting, the adversary may also exploit the structure among objects to infer their identities. For example, traces of Internet traffic often include sequential address scans made by attackers probing for vulnerable hosts. By assuming that an anonymized trace probably includes such scans, an adversary can hunt for their likely presence, such as by noting that a series of unanswered SYN packets occur close together in one part of the trace, or that (when using *sequential numbering*) suddenly a group of new hosts appears in the trace. They can then infer the original addresses of other hosts by the sequence they occupy in the scan, given the assumption that the scan started at a particular base address and proceeded sequentially up from it [104]. In addition, if the adversary has identified a single host in the trace (say a well-known server), they can then calibrate their inference by confirming that it shows up in the scan in the expected sequence.

Shared-Text Matching

When attributes or identifiers of different objects share the same text, the unmasking of one can lead to exposure of the other. For example, if there is both a user name “alice” and a file name “alice”, the user name will be exposed if the adversary can identify the file. To avoid this attack, we apply “type-separation”: the user name “alice” should be anonymized as the string “user+alice”, and the file name as “file+alice”. Generally it is good practice to avoid using the same text for distinct objects (e.g., files with the same name on different servers) unless there is some trace analysis value in doing so. The attack on prefix-preserving IP anonymization also exploits shared-text matching for cascading effects, where the shared text is the prefix.

Known-Text Matching

When both the original text and the anonymized text are known to the adversary, they can identify all appearances of the anonymized text in the trace. The knowledge required for a known-text attack is often obtained through fingerprinting.

One example is a "known server log" attack: if the adversary obtains the log of a server present in the trace, they may be able to identify the mapping between client addresses and anonymized addresses through fingerprinting, and then unmask the clients' activities on other servers. (Obtaining such logs is sometimes not difficult—for example, occasionally a query to a search engine will find them, because the logs are maintained in a publicly accessible manner.)

Another example is if the adversary can insert traffic with given strings, such as a particular user ID, into the trace, similar to the "active fingerprinting" discussed above. They can then observe how the string was mapped, and look for other occurrences of the resulting text in order to unmask instances of the same original text.

A general method to counter known-text attacks is through "knowledge separation". This is similar to the type-separation defense against shared-text matching discussed above. For example, to counter a "known server log" attack, we can anonymize a client IP differently depending on the server it accesses. To counter the user ID insertion attack, we can anonymize user IDs differently depending on whether the login is successful or not (an alternative is to anonymize user IDs depending on the client's IP address). Similarly, "active fingerprinting" with forged source IPs can defeated by anonymizing addresses differently for connections that are never established, since the adversaries will fail to complete the TCP three-way handshake unless they can conduct an initial-sequence-number guessing attack.

When we apply "knowledge separation", a single object can have multiple identifiers in the anonymized trace, which reduces the value of the trace for some types of analysis. This is a basic trade-off, and the choice of the degree to incur it will be policy-dependent.

4.5 Case Study: FTP Anonymization

In light of these possible attacks and defenses, we now turn to the anonymization scheme we used for LBNL's FTP traces. Though the scheme is inevitably dependent on the specific policy approved by the site, and thus may not be directly applicable to other sites, we believe the considerations and techniques, for instance, the “filter-in” principle, will be also applicable to other site policies and other application protocols. Accordingly, we discuss in detail relevant points of the resulting anonymization process. The full scheme can be found at [79].

The FTP traces were collected at the Internet access point (Gigabit Ethernet) of LBNL, and contain incoming anonymous FTP connections to port 21. The traces do not include any of the transferred FTP items (files uploaded or downloaded, or directory contents corresponding to the FTP “LIST” command), but only requests and replies.

As stated above, our objectives are: (1) ensure that the anonymization hides the identity of clients, non-public FTP servers, and non-public files, as well as confidential authentication information;[3], and (2) the anonymization keeps the original request/reply sequence and other nonsensitive information intact.

In some ways, these goals and the resulting traces are quite modest. But we believe that the path to site’s becoming open to releasing traces with packet contents is one that must be tread patiently, as sites quite naturally must develop a solid sense that trust in the anonymization process is warranted.

Self-Explanatory. Besides the above objectives, we designed the anonymization scheme to be *self-explanatory*: it should be easy for other people to examine and validate the scheme by merely looking at the scheme description or the policy script, without being familiar with every detail of the FTP protocol. We believe this is particularly important in order for the policy makers at a site to understand and accept trace anonymization.

4.5.1 The Filter-In Principle

The key to obtaining a robust and coherent anonymization scheme is to apply the “filter-in” principle, which is that the anonymization policy script explicitly specifies what data to leave in the clear, and everything else is anonymized (or removed). Thus, “filtering-in” implies using “white lists” of what is permitted instead of “black lists” of what is disallowed. The design choice in our framework of “explicit rewriting” also reflects the “filter-in” principle.

It is critical to employ “filter in” instead “filter out”. Anonymizing FTP traffic is complex enough that if we try to “filter out” private information by enumerating all the sensitive data fields, it is very likely that

[3]Here, hiding a “non-public” server/file means that if an adversary does not know where to find the server/file beforehand, they will not be able to find it after looking at the anonymized traces.

we will miss some of them. Also, a “filter-out” scheme would be hard to verify, unless the verifier can themselves enumerate all of the sensitive fields.

Following the “filter-in” principle, the difference between a crude anonymization script and a refined one is that the refined script will preserve more nonsensitive information in the output trace; but the two scripts should be equally privacy-safe (though we must keep in mind the maxim that complexity is the enemy of robust security). Also, a “filter-in”-style anonymization scheme is to some degree self-explanatory—verification of the scheme does not require enumerating every possibility.

4.5.2 Selected Details of FTP Anonymization

IP addresses (which appear in IP headers, PORT arguments, and some reply messages such as reply to the PASV command) are sequentially numbered, since the site views preserving client privacy as vital. (Recognizing IP addresses in reply messages is discuss in Section 4.5.4.)

User IDs (arguments of USER/ACCT commands) are anonymized except for “anonymous”, “guest”, and “ftp”. However, the anonymizer leaves a user ID in the clear if the login attempt fails and the user ID is one of the IDs defined as sensitive in `bro`’s default security policy (for example, “`backdoor`”, “`bomb`”, “`issadmin`”, “`netphrack`”, “`r00t`”, “`sync`”, “`y0uar3ownd`”, and many others). This allows us to preserve one form of attack, namely attempted backdoor access, without exposing any actual account information.

When we anonymize a user ID, we apply HMAC-MD5, annotating the user ID prior to hashing with (1) the server IP to prevent “shared-text” matching, and (2) an indication of whether the login was successful to prevent “known-text” matching.

Password. We replace the arguments of PASS commands with the string “`<password>`”. (An alternative would be to hash passwords for anonymous logins, with the email addresses annotated with the client IP address to achieve “knowledge separation”.)

File/directory names are replaced by the string “`<path>`” for non-anonymous logins. For anonymous logins, file names are left in the clear if they appear on a white list of well-known sensitive file names (e.g., “`/etc/passwd`”), in order to preserve occurrences of attacks; and anonymized with hashing otherwise. The hashing input is the absolute path annotated with the server IP to minimize shared-text matching across directories or servers. The reason to anonymize file names even for anonymous FTP traffic is that we cannot readily tell truly public files apart from private (hidden) ones that happen to be accessed using anonymous FTP, but only by users who know the otherwise unpublicized location of the file.

Arguments of commands with pre-defined argument sets (TYPE, STRU, MODE, ALLO, REST, MACB) are left intact if well-formed. For example, a TYPE argument should match the regular expression

```
/([AE](␣[NTC])?)|I|(L[0-9]+)/
```

according to RFC 959. However, the anonymizer does not assume clients follow the RFC—it checks whether the argument matches the pattern, and leaves it in the clear only if that is the case, otherwise anonymizing the argument as a string.

We apply similar techniques for the "HELP" and "SITE" commands, for which we only expose the arguments if they match a manually determined "white list" of privacy-safe HELP/SITE arguments.

Unrecognized commands are anonymized along with their arguments and recorded for optional manual inspection.

Timestamps/dates are left in the clear. While timestamps could help an adversary match up known traffic (such as traffic they injected) with its occurrence in the trace, there are enough other ways the adversary can perform such matching (by making the injected traffic singular) that leaving them intact costs little. On the other hand, timestamps are valuable for various research purposes. [4]

File sizes are considered to be safe. As argued when analyzing fingerprinting, exposing file sizes may allow the adversary to identify public files. But this is not a concern for LBNL.

Server software version/configuration is also considered to be safe, as the information that can be inferred from the trace can be readily obtained through other means (since the servers are public).

4.5.3 Refining with Manual Inspection

Whether data is to be left in the clear or anonymized, the anonymous script logs the decision and the reason for later inspection.[5] Identical entries are only logged once. Inspection of the log (with various text

[4]We chose to preserve timestamps in the clear when we devised this FTP anonymization scheme in 2003. However, two years later researchers discovered that hosts can be fingerprinted with TCP timestamp options[57], if there are a sufficient number of timestamps in the trace, as each physical device has its unique clock skew. While this particular attack poses little threat to old traces, such development highlights the devilish nature of anonymization.

[5]Here we assume that the administrator of the trace anonymization can see the original trace—this helps in verifying results and generating better traces.

processing tools) helps us to discover (1) privacy holes (or to demonstrate the absence of holes), and also (2) overly conservative anonymization of nonsensitive information (important for working towards more refined scripts). We discuss log inspection techniques in detail below.

A "filter-in"-style script always makes conservative judgments on unknown data. Sometimes it can be too conservative, missing an opportunity to expose interesting, nonsensitive data, e.g., a mistyped command like "UUSER" or a user id like "annonymous". It is difficult to hardwire such commands and user names into the general anonymization script, as they may appear in unpredictable forms. Nevertheless, these special cases do not appear very often in traces, so we can afford to *manually inspect* each case by looking at the log after anonymization and then customizing the script to expose the nonsensitive ones. Figure 4.4 shows three log entries we have seen: the first entry records a common-case anonymization of a path name; while the other two, recording anonymizations of the "UUSER" command and user name "annonymous", are the kinds of entries we look for during manual inspection.

Note that the customization for special cases should be *optional*. The script should always first anonymize any unknown data, and should make no assumptions about whether the log will be manually inspected.

As most entries in the anonymization log record the anonymization of "common" cases, the trick to digging up special cases is to look for deviant entries through *text classification*. Here, we examine command arguments as an example to illustrate how we discover special cases:

First, we classify entries by the type of data being anonymized. The type can be, for example, a non-guest user name (e.g., the misspelled "annonymous"), or a non-public file name, or the argument of a PORT command. Some types of anonymization, e.g., of path names and passwords, happen very often, while others rarely appear in the log. These rare types of anonymization often present interesting cases. For example, for a trace of an FTP server that only allows anonymous login, there can still be a few user names being anonymized. We have seen: "`anno`", "`anonyo\010`", "`anonymouse`", "`help`", and "`anamouse`", as well as a password mistyped for a `USER` command. Except for the password, all of the other user names actually do not reveal any private information. *But it's important to catch the password.* Note that none of these strange user names will appear in the output trace unless we modify the script to explicitly allow them, so the password will not appear without specific action to keep it.

Furthermore, we look for "malformed" path names—those do not match a heuristic pattern for well-formed path names. We find, for example: "`#`", "`\xd0\xc2\xce\xc4\xbc\xfe\xbc\xd0`", "`/n/n`

```
anonymize_arg: (path name) [CWD] "conferencing" to "U42117b96U" in [x.x.x.x/x > x.x.x.x/ftp]
anonymize_cmd: (unrecognized command) "UUSER" [anonymous] to "U7b402a69U" in [x.x.x.x/x > x.x.x.x/ftp]
anonymize_arg: (user name) [USER] "annonymous" to "Ufb6db9afU" in [x.x.x.x/x > x.x.x.x/ftp]
```

Figure 4.4: Anonymization Log Entries

`This file was not retrieved by Teleport Pro, because it did not meet the project`”.[6]

In addition, applying similar techniques lets us find misspelled commands, or commands containing control characters: e.g., “`USE`”, “`UUSER`”, “`RETR<BS><BS><BS><BS>`”, all of which we have seen in practice. These commands likely indicate users typing directly rather than using client software, therefore it is valuable to preserve this information.

4.5.4 Reply Anonymization

An FTP reply consists of a reply code and a text message. We leave reply codes in the clear, as they do not reveal any private information. Reply messages, on the other hand, do often contain sensitive information and are hard to anonymize because there is no standard format for most reply messages—the format depends on the server implementation and its configuration.

One possibility is to discard the original text (except for replies to PASV, which are well-defined) and replace it with a dummy message. This has the virtue of being simple. On the other hand, reply messages do sometimes carry useful information that cannot be inferred from the reply codes. For example, a reply of code 530 (denial of login) usually explains why the login was rejected–it can be “guest login not permitted” or “Sorry, the maximum number of users from your host are already connected”. Such information can be valuable in some cases. So we explored methods to anonymize FTP replies.

As messages may contain variables such as file names/sizes, dates, and domain names, there can be countless distinct messages. However, we observe that there is only a limited set of message *templates*, as the number of templates is bounded by the number of different server software/configurations at the site. And we can extract templates (along with human assistance) by comparing messages against each other and distilling the common parts. Figure 4.5 shows a few example message templates. Once we have extracted the message templates, we can parse messages by matching them against the templates and thereby understanding the semantics of the data elements in the text.

[6]Teleport Pro is the name of an offline browser.

```
150 |opening| |ascii, binary| |mode| |data| |connection| |for| |˜ arg| |˜ ip| |˜ num| |˜ num| |bytes|
211 |connected| |to| |˜ domain, ˜ ip|
220 |welcome| |to| |˜ *| |ftp| |server|
550 |˜ arg| |not| |a| |directory|
```

Figure 4.5: FTP Reply Message Templates

message:	`"150 Opening BINARY mode data connection for /def.pdf (123.45.67.89,50034) (156678 bytes)"`
split →	`"150 |opening| |binary| |mode| |data| |connection| |for| |/def.pdf| |123.45.67.89| |50034| |156678| |bytes|"`
abstract →	`"150 |opening| |binary| |mode| |data| |connection| |for| |˜ arg| |˜ ip| |˜ num| |˜ num| |bytes|"`
merge →	`"150 |opening| |ascii, binary| |mode| |data| |connection| |for| |˜ arg| |˜ ip| |˜ num| |˜ num| |bytes|"`

Figure 4.6: Message Template Extraction

Message templates are first automatically extracted by a script then manually sanitized before being used for template matching. The automated template extraction is done in three steps: splitting, abstraction, and merging (as shown in Figure 4.6). We first *split* a message into parts—each part contains a word or a data element such as an IP address or a file name. Next, in *abstraction*, we try to guess whether each part is a variable or a constant part of the message template. Through *abstraction* we are able to find most of variable slots in message templates, and *merging* helps to reveal the rest of them. We merge two templates when they are identical on all but one part, and this process is iterated till no templates can be further merged.

The message extraction process is refined through the accumulation of experience. We found that the key issue in abstraction is to recognize the corresponding command argument echoed in the reply message. This is tricky because the echoed argument is sometimes different from the original argument, particularly when it is a file name. For example, the echoed argument can be the absolute file path or only contain the base file name with the directory parts. Therefore we need to recognize variants of the argument. The key for good message splitting is to know where *not* to split. By default we split at spaces and punctuation; however, we do not want to split an IP address or a file name, otherwise they cannot be recognized during abstraction.

Extracted message templates need to be examined and sanitized before being used for message matching. This can be a tedious process and we strived to minimize the required effort. Currently, when extracting templates from a set of ten-day long FTP traces, which contain more than 1.4 M lines of replies in 22.6 K connections to 318 distinct servers, we wound up with 461 message templates for 32 kinds of reply codes. Among the 461 templates, 25 require sanitization to remove server identity information. Examining a few hundred templates is feasible but still not easy—perhaps this is the price for processing free format text.

4.5.5 Verification

Verification is a fundamental step of the anonymization process. No matter how much thought we apply to the anonymization policy, the safety of the anonymization also depends on the correctness of the policy script and on the underlying `bro` mechanisms. Therefore, besides inspecting the anonymization description and script, it is also important to examine the output trace directly.

Ideally, the verification process would guarantee that the transformed trace complies with the *intended* anonymization policy. This is a different notion that the *expressed* anonymization policy, due to the possiblity of errors occurring in coding up the expression. Our strategy therefore is to attempt to analyze the general properties of the transformed trace without tying these too closely to the anonymization script that was used to effect the transformation. As such, we cannot guarantee that there are no "holes" in the anonymized trace (and indeed doing so appears fundamentally intractable). Instead, we aim to provide another level of precaution. In general, it is particularly important to have a strong "verification story" in order to persuade sites that the anonymization process will meet their requirements.

For verification we do not use `bro` to parse the output trace's packets—doing so would introduce a common point of failure across anonymization and verification. Instead, we look at the packets directly, using different tools. Automating the verification process remains an open problem—currently, it requires human assistance, although some of the steps can be automated to reduce the burden.

For packet headers, we inspect the source and destination IP addresses. As the anonymized addresses are sequentially numbered, verification that these lie in the expected range can be performed automatically.

For FTP requests in packet payloads, we enumerate all distinct commands and arguments present in the trace, except those which are already hashed (hash results follows a particular textual format and thus can automatically excluded). When the text parts of reply messages are discarded, it is straightforward to verify that FTP replies only contain reply codes and a placeholder of dummy text.

When we choose to anonymize reply messages, verification consists of two parts, checking vocabulary and numbers, respectively. Vocabulary checking is similar to message template extraction, but simpler and implemented separately. Messages are again split at blanks and punctuation, this time without worrying about special cases as in splitting for message template extraction. Next we abstract the parts by two rules: (1) if a part is a decimal number, substitute it with the string "`<num>`"; (2) if a part is a hashing output, substitute it with the string "`<hash>`". This way we can reduce 1.4 M anonymized messages to about

600 patterns. We then manually inspect these, which can be expedited by first sorting them so that similar patterns are clustered.

In checking numbers we are mainly concerned about numbers constituting IP addresses. Accordingly, we look for any four consecutive number parts in split messages and record each instance that does not fall within the range of anonymized addresses. Interestingly, such cases *do* appear, though they are quite rare, and safe—e.g., part of a software version string such as "wu-2.6.2(1)".

Verification helped us find a potential hole in an earlier version of our anonymization script. We found two suspicious command arguments: "`GSSAPI`" and "`KERBEROS_V4`". Though the strings themselves do not disclose any private information, their appearance is alarming because they are not defined anywhere to be "safe" in the script.

Looking into the logs revealed that they were arguments for two rejected "AUTH" commands. According to RFC 2228, the argument for the "AUTH" command specifies the authentication mechanism. Thus, a rejected mechanism seems safe to expose. However, doing so overlooks the possibility that a user might mistakenly specify sensitive information, such as a password, instead of an authentication *mechanism*. A "fail-safe" solution is to white list "`GSSAPI`" and "`KERBEROS_V4`" and anonymize any unknown argument for the "AUTH" command.

4.5.6 Discussion

Integrity of Output Trace. Besides the absence of private information, we also want to check whether the packets, TCP flows, and FTP requests and replies in the anonymized trace are all *well-formed*. To do so, we run `bro`'s FTP analyzer on the anonymized traces to see whether `bro` can reassemble the TCP flows and parse the FTP requests and replies. We compare the FTP logs from both traces. `bro`'s FTP log records start and finish of FTP sessions and all requests and replies in the session. For a day-long FTP trace of 80 MB, 8,871 connections, and 86,908 request-reply pairs, we find that the two logs have the same FTP session starting timestamps,[7] request command sequences (not including the arguments) and reply code sequences, also at the same timestamps. For command arguments and reply messages, we cannot compare them directly as of course many of them are anonymized. We randomly picked a few sessions and manually checked the arguments and messages.

[7]In some cases, `bro`'s connection termination is triggered by a timer, which results in slightly different session finish timestamps.

FTP analyzer	131 seconds
FTP analyzer + anonymizer	1009 seconds
FTP analyzer + dummy rewriter	192 seconds

Figure 4.7: Execution time of various FTP policy scripts

Anonymized Traces for Intrusion Detection. As mentioned earlier, packet traces are particularly valuable for research on network intrusion detection. So we very much want trace anonymization to preserve intrusion-like activities. This applies both to preserving actual attacks, but, even more so, unusual-but-benign traffic that stresses the false-positive/false-negative accuracy of intrusion detection algorithms. This latter is particularly important because it is often a key element missing from assessments of network intrusion detection mechanisms—it is easy for researchers to attain traces of actual attacks, because they can generate these using the plethora of available attack tools, but it is much more difficult today for researchers to attain detailed traces of background traffic.

Generally whether an attack survives anonymization depends on both its characteristics and how it is detected. Some FTP intrusions are recognized by signatures of files or user IDs the intruder tries to access or login as. For example, directory name "`tagged`" is often associated with FTP warez attacks; failed "`root`" or "`sysadm`" login attempts suggest server backdoor probing. Preserving these attacks requires leaving relevant identifiers in the clear. Fortunately the identifiers are mostly well-known and do not expose private identities, so they can kept through anonymization by establishing a white list for "sensitive" file names and user IDs to leave in the clear. To do so, however, requires knowing the attack signatures beforehand; thus, attacks with unknown signatures may still be lost in anonymization.

Other types of intrusions are recognized by activity patterns rather than identifier signatures. Most of these attacks can survive anonymization. For instance, port scanning is marked by unanswered (or responded by TCP-RST) TCP-SYN packets from the same source host to different destination hosts; successive failed attempts at creating directories on multiple servers may imply an FTP warez attack.

Performance. Figure 4.7 shows the CPU time spent on a 1 GHz Pentium III processor running on the day-long trace mentioned above. We see that the FTP anonymizer, which also requires the FTP analyzer, is 7.7 times slower than the FTP analyzer. To understand where time is spent, we also tested `bro` with a dummy FTP trace rewriter, which simply writes the original requests and replies to the output trace. We find that the execution overhead of the anonymizer script itself heavily dominates, comprising 81% of the total processing. The time is spent performing numerous hash table lookups, string operations, and regular expression matches, and generating a 3.8 MB anonymization log. We find this performance adequate, especially for off-line anonymization. It even suffices for on-line anonymization for FTP, though when extended to a higher volume protocol such as HTTP may prove problematic.[8]

4.6 Related Work

TCPdpriv [66] anonymizes `tcpdump` traces by stripping packet contents and rewriting packet header fields. One of its features is a form of "prefix-preserving" anonymization of IP addresses (the "-A50" option). [135] analyzes the security implications of this anonymization, proposing an approach that might be used to crack the "-A50" encoding by first identifying hosts with well-known traffic patterns (e.g., DNS servers). Xu et al proposed a cryptography-based scheme for prefix-preserving address anonymization [132]. The scheme can maintain a consistent anonymization mapping across multiple anonymizers using a shared cryptographic key. Peuhkuri presented an analysis of the private information contained in TCP/IP header fields and proposed a scheme to anonymize packet traces and store the results in a compressed format [91]. Peuhkuri's scheme for network addresses anonymization cannot be directly applied to our work because the scheme generates 96 bits instead of 32 bits for each address, and we are constrained by needing to generate output in `tcpdump` format. Finally in recent work with colleagues [78] I explored the devilish issues in anonymizing traces collected inside an enterprise network. All of these works address only the anonymization of TCP/IP headers, with no mechanisms for retaining packet payloads.

NetDuDe (NETwork DUmp data Displayer and Editor) [58] is a GUI-based tool for interactive editing of packets in `tcpdump` trace files. NetDuDe itself does not parse application-level protocols, but allows user to write plug-in's for packet processing, e.g., a checksum fixer plug-in can recompute checksums and update the checksum fields in TCP and IP headers.

There has also been considerable work on extracting application-level data from online traffic, though without significant applications to content-preserving anonymization. Gribble et al built an HTTP parser to extract HTTP information from a network sniffer [41]. Feldmann in [31] describes BLT, a tool to extract complete HTTP headers from high-volume traffic, and discusses various challenges in extracting accurate HTTP fields. Pandora [84] is a component-based framework for monitoring network events, which contains, among others, components to reconstruct HTTP data from packets. It is similar in spirit

[8]Note that the HTTP rewriter used to reduce HTTP packet traces as discussed in Section 4.3.4 runs on-line, processing nearly 100 times the daily data volume, though in a simpler fashion.

to Windmill [64]. Ethereal is able to reconstruct TCP session streams, and parses the stream to extract application protocol level data fields [29]. The fields can be used to filter the view of the trace. Ethereal has a GUI-based interface to display trace data. There are also numerous commercial network monitoring systems that can extract application-level information, e.g., EtherPeek[128].

There are also efforts on setting up honeypots [45] and break-in challenges [18] to collect traces of network intrusions. Such pure intrusion traces have the virtue of containing little private information, as the target hosts are not used for other purposes. For the same reason, however, the traces do not contain background traffic with various unusual-but-benign activities, and thus are very different from traffic at an operational site.

Finally, Mogul argues "Trace Anonymization Misses the Point" [67], proposing an alternative strategy to trace anonymization—instead of sharing anonymized traces, researchers send reduction agents to the site that has the source trace data. We believe our tool is in fact complementary to this sort of approach. Mogul raises the question: what kind of code should be sent to the source sites? Our answer is: "a `bro` script for trace transformation."

4.7 Summary

In this work we have designed and implemented a new tool for packet trace anonymization and general purpose transformation. The tool offers a great degree of freedom and convenience for trace transformation by providing a high-level programming environment in which transformation scripts operate on application-level data elements.

Using this framework, we developed an anonymization script for FTP traces and applied it to anonymizing traces from LBNL for public release. Unlike previous packet trace anonymization efforts, packet payload contents are included in the result. We discussed the key anonymization principle of "filter-in" as opposed to "filter-out", and the crucial problem of *verifying* the correctness of the anonymization procedure. We also analyzed a class of inference attacks and how we might defend against them.

We believe this tool offers a significant step forward towards ending the current state of there being *no* publicly available packet traces with application contents. As such, we hope to help open up new opportunities in Internet measurement and network intrusion detection research.

Chapter 5

Characteristics of Internet Background Radiation

Monitoring any portion of the Internet address space reveals incessant activity. This holds even when monitoring traffic sent to unused addresses—thus we term the traffic "background radiation." Background radiation reflects fundamentally nonproductive traffic, either malicious (flooding backscatter, scans for vulnerabilities, worms) or benign (misconfigurations). While the general presence of background radiation is well known to the network operator community, its nature had not been previously characterized. This thesis develops a broad characterization based on data collected from unused networks in the Internet. Three key elements of the methodology are (1) the use of filtering to reduce load on the measurement system, (2) the use of active responders to elicit further activity from scanners in order to differentiate different types of background radiation, and (3) the use of application level traffic semantic analysis to uncover activity details at application protocol level. This study breaks down the components of background radiation by protocol, application, and often specific exploit; analyzes temporal patterns and correlated activity; and assesses variations across different networks and over time. While a menagerie of activity is found in background radiation, probes from worms and autorooters heavily dominate the traffic.

This chapter proceeds as follows. Section 5.1 defines "Internet background radiation" and the goals of this study. Section 5.2 discusses related work. Section 5.3 describes the sources of data used in this study and the methodology related to capturing and analyzing this data. Section 5.4 analyzes what we can learn from our monitoring when we use it purely passively, and Section 5.5 then extends this to what we can learn if we also respond to traffic we receive. In Section 5.6 we evaluate aspects of traffic source behavior. We conclude with a summary of our study in Section 5.7.

5.1 Problem Statement

In recent years a basic characteristic of Internet traffic has changed. Older traffic studies make no mention of the presence of appreciable, on-going attack traffic [24, 87, 116, 6], but those monitoring and operating today's networks are immediately familiar with the incessant presence of traffic that is "up to no good." We can broadly characterize this traffic as *nonproductive*: it is either destined for addresses that do not exist, servers that are not running, or servers that do not want to receive the traffic. It can be a hostile reconnaissance scan, "backscatter" from a flooding attack victimizing someone else, spam, or an exploit attempt.

The volume of this traffic is not minor. For example, traffic logs from the Lawrence Berkeley National Laboratory (LBL) for an arbitrarily-chosen day in 2004 show that 138 different remote hosts each scanned 25,000 or more LBL addresses, for a total of about 8 million connection attempts. This is more than double the site's entire quantity of successfully-established incoming connections, originated by 47,000 distinct remote hosts. A more fine-grained study of remote scanning activity found (for a different day) 13,000 different scanners probing LBL addresses [51].

What is all this nonproductive traffic trying to do? How can we classify various types of activity in order to detect *new* types of malicious activity?

Because this new phenomenon of incessant nonproductive traffic has not yet seen detailed characterization in the literature, we have lacked the means to answer these questions. This study aims to provide an initial characterization of this traffic. Given the traffic's pervasive nature (as we will demonstrate), we term it Internet "background radiation".

A basic issue when attempting to measure background radiation is how, in the large, to determine which observed traffic is indeed unwanted. If we simply include all unsuccessful connection attempts, then we will conflate truly unwanted traffic with traffic representing benign, transient failures, such as accesses to Web servers that are usually running but happen to be off-line during the measurement period.

By instead only measuring traffic sent to hosts that *don't exist*—*i.e.,* Internet addresses that are either unallocated or at least unused—we can eliminate most forms of benign failures and focus on traffic highly likely to reflect unwanted activity. In addition, analyzing unused addresses yields a second, *major* measurement benefit: we can safely *respond* to the traffic we receive. This gives us the means to not only passively measure unwanted traffic (for example, what ports get probed), but to then engage the remote sources in order to elicit from them their particular intentions (for example, what specific actions they will take if duped into thinking they have found a running server).

Given the newness of this type of Internet measurement, one of the contributions of our study is the set of methodologies we develop for our analysis. These include considerations for how to use *filtering* to reduce the load on the measurement system, how to construct *active responders* to differentiate different types of background radiation, and ways for interpreting which facets of the collected data merit investigation and which do not.

5.2 Related Work

Several studies have characterized specific types of malicious traffic. Moore *et al.* investigate the prevalence of denial-of-service attacks in the Internet using "backscatter analysis" [73], *i.e.,* observing not the attack traffic itself but the replies to it sent by the flooding victim, which are routed throughout the Internet due to the attacker's use of spoofed source addresses. Measurement studies of the Code Red I/II worm outbreaks [71], the Sapphire/Slammer worm outbreak [70, 69], and the Witty worm [60] provide detail on the method, speed and effects of each worm's propagation through the Internet. Additional studies assess the speed at which counter-measures would have to be deployed to inhibit the spread of similar worms [72].

The empirical components of these studies were based largely on data collected at "network telescopes" (see below) similar to those used in our study, though without an active-response component. A related paper by Staniford *et al.* mathematically models the spread of Code Red I and considers threats posed by potential future worms [112]. A small scale study of Internet attack processes using a fixed honeypot setup is provided in [23]. Yegneswaran *et al.* explore the statistical characteristics of Internet attack and intrusion activity from a global perspective [134]. That work was based on the aggregation and analysis of firewall and intrusion detection logs collected by Dshield.org over a period of months. The coarse-grained nature of that data precluded an assessment of attacks beyond attribution to specific ports. Finally, Yegneswaran *et al.* provide a limited case study in [133] that demonstrates the potential of network telescopes to provide a broad perspective on Internet attack activity. We extend that work by developing a much more comprehensive analysis of attack activity.

Unused IP address space has become an important source of information on intrusion and attack activity. Measurement systems deployed on unused IP address ranges have been referred to as “Internet Sink-holes” [40], and “Network Telescopes” [68]. Active projects focused on unused address space monitoring include Honeynet [44], Collapsar [50], Potemkin [118], Honeyd [97, 96], and GQ [21]. Honeynet, Collapsar, and Potemkin focus on the use of live virtual-machine-based systems to monitor unused addresses. Honeyd uses a set of stateful virtual responders to operate as an interactive honeypot. GQ attempts to use a combination of a RolePlayer-based [22] simulator and a virtual machine system to achieve both scalability and high interactivity.

Finally, network intrusion detection systems, including Snort [100, 101], Bro [89], and a variety of commercial tools, are commonly used to detect scans for specific malicious payloads. An emerging area of research is in the automated generation of attack signatures. For example, Honeycomb [59] is an extension of Honeyd that uses a *longest common substring* (LCS) algorithm on packet-level data recorded by Honeyd to automatically generate signatures. Other recent work pursues a similar approach, including Earlybird [109] and Autograph [54]. Our study can inform future developments of such systems with respect to both the type and volume of ambient background attack activity.

5.3 Measurement Methodology

This section describes the methods and tools we use to measure and analyze background radiation traffic, addressing three key issues:

1. **Creating deep conversations**: We find that `TCP/SYN` packets dominate background radiation traffic in our passive measurements, which means we need to accept connections from the sources and extend the dialog as long as possible to distinguish among the types of activities. The key problem here is building responders for various application protocols, such as HTTP, NetBIOS, and CIFS/SMB, among others.

2. **Taming large traffic volume**: We listen and respond to background traffic on thousands to millions of IP addresses. The sheer volume of traffic presents a major hurdle. We handle this with

two approaches: 1) devising a sound and effective filtering scheme, so that we can significantly reduce the traffic volume while maintaining the variety of traffic; and 2) building a scalable responder framework, so we can respond to traffic at a high rate.

3. **Analyzing traffic semantics**: We capture network traffic between radiation sources and our responders and analyze application level semantics of various interaction.

5.3.1 Application-Level Responders

Our approach to building responders was “data driven”: we determined which responders to build based on observed traffic volumes. Our general strategy was to pick the most common form of traffic, build a responder for it detailed enough to differentiate the traffic into specific types of activity, and once the “Unknown” category for that type of activity was sufficiently small, repeat the process with the next largest type of traffic.

Using this process, we built an array of responders for the following protocols (Figure 5.1): HTTP (port 80), NetBIOS (port 137/139), CIFS/SMB [19] (port 139/445), DCE/RPC [25] (port 135/1025 and CIFS named pipes), and Dameware (port 6129). We also built responders to emulate the backdoors installed by MyDoom (port 3127) and Beagle (port 2745) [13], [74].

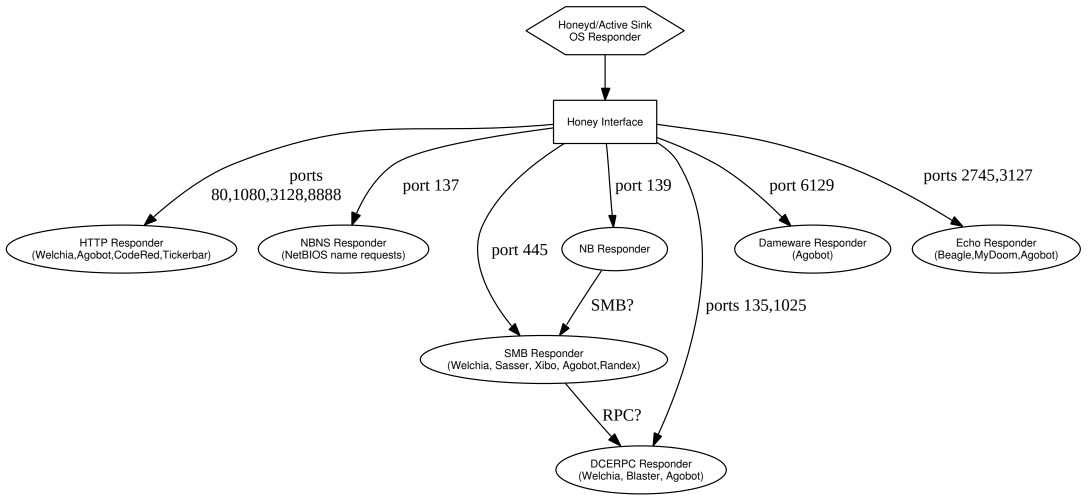


Figure 5.1: Top level Umbrella of Application Responders

Application-level responders need to not only adhere to the structure of the underlying protocol, but also to know *what* to say. Most sources are probing for a particular implementation of a given protocol, and we need to emulate behavior of the target software in order to keep the conversation going.

The following example of HTTP/WebDAV demonstrates what this entails. We see frequent `"GET /"` requests on port 80. Only by responding to them and mimicking a Microsoft IIS with WebDAV enabled will we elicit further traffic from the sources. The full sequence—in which the "411 Length Required" response indicates that WebDAV is enabled, which then attracts the attack—plays as:

```
GET /
⇒ |200 OK ... Server: Microsoft-IIS/5.0|
SEARCH /
⇒ |411 Length Required|
SEARCH /AAA... (URI length > 30KB)
⇒ (buffer overflow exploit received)
```

Some types of activity require quite intricate responders. Many Microsoft Windows services run on top of CIFS (port 139/445), which lead us to develop the detailed set of responses shown in Figure 5.2. Requests on named pipes are further tunneled to various DCE/RPC responders. One of the most complicated activities is the exploit on the SAMR ("Security Account Manager Remote") and later on the SRVSVC ("Server Service") pipe, which involves more than ten rounds exchanging messages before the source will reveal its specific intent by attempting to create an executable file on the destination host. Figure 5.3 shows an example where we cannot classify the source until the "NT Create AndX" request for `msmsgri32.exe`. (The NetrRemoteTOD command is used to schedule the worm process to be invoked one minute after TimeOfDay [11].) We found this attack sequence is shared across several viruses, including the Lioten worm [11] and Agobot variants [2].

Building responders like this one can prove difficult due to the lack of detailed documentation on services such as CIFS and DCE/RPC. Thus, we sometimes must resort to probing an actual Windows system running in a virtual machine environment in order to analyze the responses it makes en route to becoming infected. We modified existing trace replay tools like `flowreplay` for this purpose [36].

More generally, as new types of activity emerge over time, our responders also need to evolve. While we find the current pace of maintaining the responders tractable, an important question is to what degree we can automate the development process.

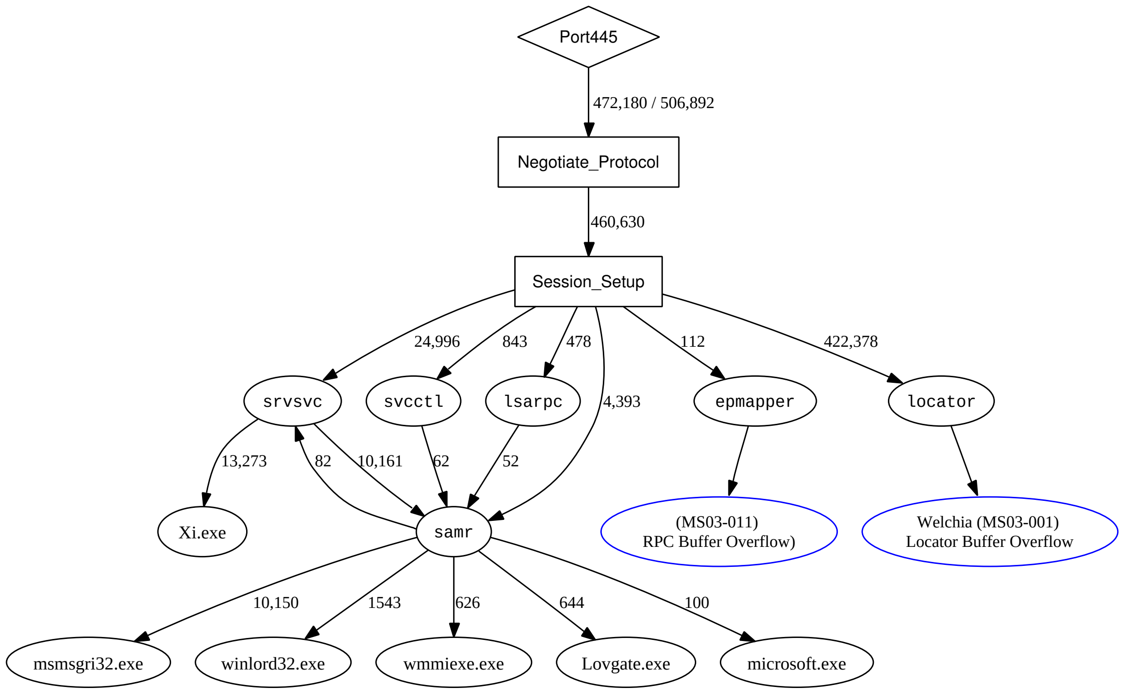


Figure 5.2: Example summary of port 445 activity on Class A. 506,892 sessions in total. Arcs indicate number of sessions.

```
-> SMB Negotiate Protocol Request
<- SMB Negotiate Protocol Response
-> SMB Session Setup AndX Request
<- SMB Session Setup AndX Response
-> SMB Tree Connect AndX Request,
        Path:  \\XX.128.18.16\IPC$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request, Path: \samr
<- SMB NT Create AndX Response
-> DCERPC Bind: call_id: 1 UUID: SAMR
<- DCERPC Bind_ack:
-> SAMR Connect4 request
<- SAMR Connect4 reply
-> SAMR EnumDomains request
<- SAMR EnumDomains reply
-> SAMR LookupDomain request
<- SAMR LookupDomain reply
-> SAMR OpenDomain request
<- SAMR OpenDomain reply
-> SAMR EnumDomainUsers request
```

Now start another session, connect to the SRVSVC pipe and issue NetRemoteTOD (get remote Time of Day) request

```
-> SMB Negotiate Protocol Request
<- SMB Negotiate Protocol Response
-> SMB Session Setup AndX Request
<- SMB Session Setup AndX Response
-> SMB Tree Connect AndX Request,
        Path: \ \XX.128.18.16\IPC$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request, Path: \srvsvc
<- SMB NT Create AndX Response
-> DCERPC Bind: call_id: 1 UUID: SRVSVC
<- DCERPC Bind_ack: call_id: 1
-> SRVSVC NetrRemoteTOD request
<- SRVSVC NetrRemoteTOD reply
-> SMB Close request
<- SMB Close Response
```

Now connect to the ADMIN share and write the file

```
-> SMB Tree Connect AndX Request, Path: \\XX.128.18.16\ADMIN$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request,
    Path:\system32\msmsgri32.exe  <<<===

<- SMB NT Create AndX Response, FID: 0x74ca
-> SMB Transaction2 Request SET_FILE_INFORMATION
<- SMB Transaction2 Response SET_FILE_INFORMATION
-> SMB Transaction2 Request QUERY_FS_INFORMATION
<- SMB Transaction2 Response QUERY_FS_INFORMATION
-> SMB Write Request
....
```

Figure 5.3: Active response sequence for Samr-exe viruses

5.3.2 Taming the Traffic Volume

Responding to the entirety of background radiation traffic received by thousands to millions of IP addresses would entail processing an enormous volume of traffic. For example, we see nearly 30,000 packets per second of background radiation on the Class A network we monitor. Taming the traffic volume requires effective filtering and a scalable approach to building responders. We discuss each in turn.

Filtering

When devising a filtering scheme, we try to balance trade-offs between traffic reduction and the amount of information lost in filtering. [1] We considered the following strategies:

Source-Connection Filtering: This strategy keeps the first N connections initiated by each source and discards the remainder. A disadvantage of this strategy is that it provides an inconsistent view of the network to the source: that is, live IP addresses become unreachable. Another problem is that an effective value of N can be service- or attack-dependent. For certain attacks (*e.g.*, "Code Red"), $N = 1$ suffices, but multi-stage activities like Welchia, or multi-vector activities like Agobot, require larger values of N. Moreover, if a source tries to contact more than N destination addresses at the same time, our view will be limited to at most one connection per source-destination pair.

Source-Port Filtering: This strategy is similar except we keep N connections for each source/destination port pair. This alleviates the problem of estimating N for multi-vector activities like Agobot, but multi-stage activities on a single destination port like Welchia remain a problem. This strategy also exposes an inconsistent view of the network.

Source-Payload Filtering: This strategy keeps one instance of each type of activity per source. From a data richness perspective, this seems quite attractive. However, it is very hard to implement in practice as we do not often know whether two activities are similar until we respond to several packets (especially true for multi-stage activities and chatty protocols like NetBIOS). This strategy also requires significant state.

Source-Destination Filtering: This is the strategy we chose for our experiments, based on the assumption that background radiation sources possess the same degree of affinity to all monitored IP addresses.

[1]Filtering is mostly the work of my collaborators, Vinod Yegneswaran and Vern Paxson, and is included in this dissertation to preserve completeness of the study.

More specifically, if a source contacts a destination IP address displaying certain activity, we assume that we will see the same kind of activity on all other IP addresses that the source tries to contact. We find this assumption generally holds, except for the case of certain multi-vector worms that pick one exploit per IP address, for which we will identify only one of the attack vectors.

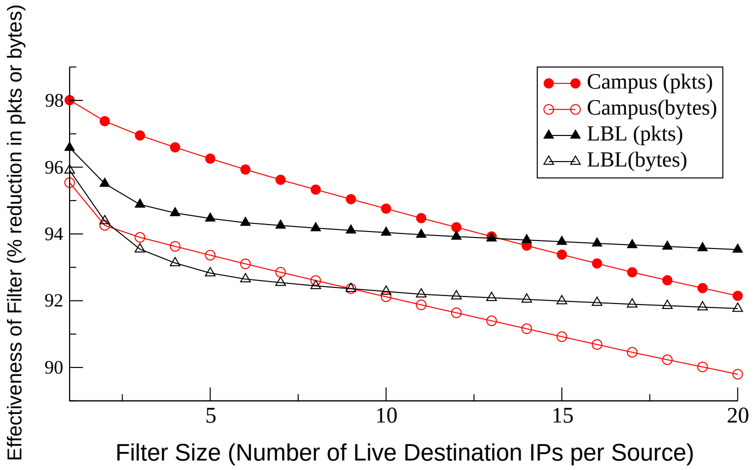
Figure 5.4 illustrates the effectiveness of this filtering on different networks and services when run for a two-hour interval. The first plot shows that the filter reduces the inbound traffic by almost two orders of magnitude in both networks. The LBL network obtains more significant gains than the larger Campus networks because the Campus network intentionally does not respond to the last stage of exploits from certain frequently-seen Welchia variants that in their last step send a large attack payload ($>$ 30KB buffer overflow). The second plot illustrates the effectiveness of the filter for the various services. Since Blaster (port 135) and MyDoom (port 3127) scanners tend to horizontally sweep IP subnets, they lead to significant gains from filtering, while less energetic HTTP and NetBIOS scanners need to be nipped in the bud (low N) to have much benefit.

With source-destination filtering, a Honeyd responder running on a single computer can easily respond to background radiation on 10 /24 subnets (2560 IP addresses). Responding on telescopes that are magnitudes larger, e.g. on a /16 network (65536 addresses), however, requires a more scalable responder platform.

Active Sink: an Event-driven Stateless Responder Platform

Part of our active response framework explores a *stateless* approach to generating responses, with a goal of devising a highly scalable architecture. [2] Active Sink is the active response component of iSink, a measurement system developed by Yegneswaran et al. [133] to scalably monitor background radiation observed in large IP address blocks. Active Sink simulates virtual machines at the network level, much like Honeyd [96], but to maximize scalability it is implemented in a stateless fashion as a Click kernel module [133] [55]. It achieves statelessness by using the form of incoming application traffic to determine an appropriate response (including appropriate sequence numbers), without maintaining any transport or application level state. A key question for this approach is whether all necessary responders can be constructed in such a stateless fashion. While exploring this issue is beyond the scope of the present work, we

[2] Active Sink is mostly the work of my collaborators, Vinod Yegneswaran and Paul Barford, and is included in this dissertation to preserve completeness of the study.



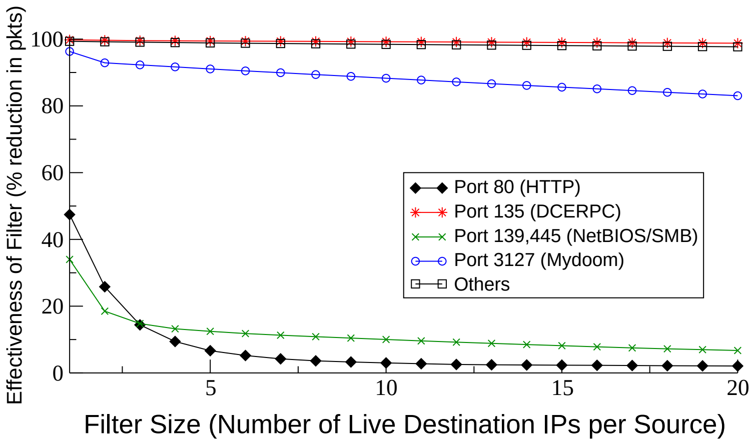


Figure 5.4: Effectiveness of Filtering, Networks (top) and Services (bottom)

note that for all of the responders we discuss, we were able to implement a stateless form for Active Sink, as well as a stateful form based on Honeyd. (To facilitate the dual development, we developed interface modules so that each could use the same underlying code for the responders.)

5.3.3 Traffic Analysis

Once we can engage in conversations with background radiation sources, we then need to undertake the task of understanding the traffic.

Here our approach has two components: first, we separate traffic analysis from the responders themselves; second, we try to analyze the traffic in terms of its application-level semantics.

It might appear that the job of traffic analysis can be done by the responders, since the responders need to understand the traffic anyway. However we believe that there are significant benefits to performing traffic analysis independently—by capturing and storing `tcpdump` packet traces for later off-line analysis. Independent analysis allows us to preserve the complete information about the traffic and evolve our analysis algorithms over time. The flip side is that doing so poses a challenge for the analysis tool, since it needs to do TCP stream reassembly and application-protocol parsing. To address this issue, we built our tool on top of the Bro intrusion detection system [89], which provides a powerful platform for application-level protocol analysis.

Our analysis has an important limitation: we do not attempt to understand the binary code contained in buffer-overrun exploits. This means we cannot tell for sure which worm or autorooter sent us a particular exploit (also due to lack of a publicly available database of worm/virus/autorooter packet traces). If a new variant of an existing worm arises that exploits the same vulnerability, we may not be able to discern the difference. However, the analysis will identify a new worm if it exploits a different vulnerability, as in the case of the Sasser worm [103].

5.3.4 Experimental Setup

We conducted our experiments at two different sites. These ran two different systems, *iSink* and *LBL Sink*, which conducted the same forms of application response but used different underlying mechanisms.

iSink: Our iSink instance monitored background traffic observed in a Class A network (/8, 2^{24} addresses), and two $/19$ subnets (16K addresses) on two adjacent UW campus class B networks, respectively.

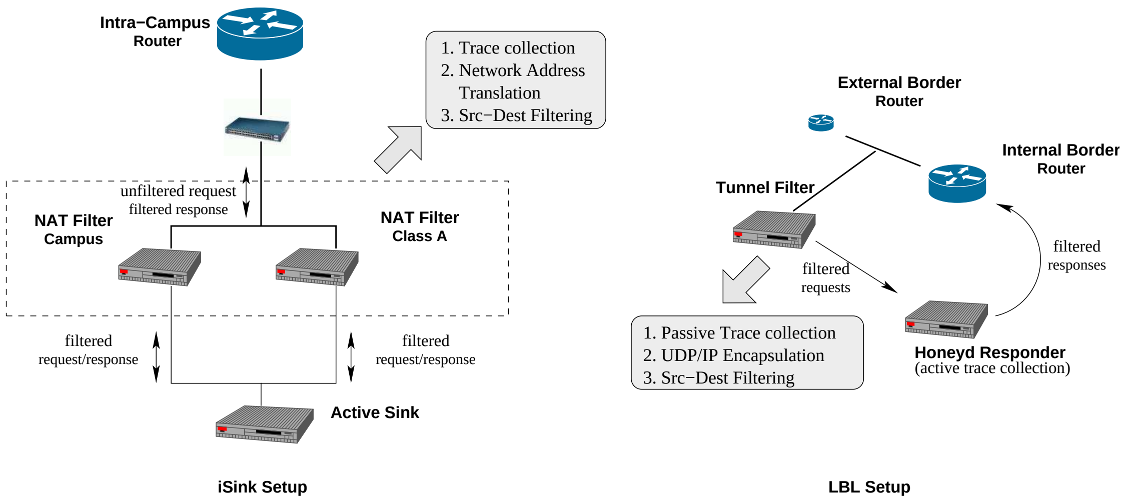


Figure 5.5: The Honeynet architecture at iSink and LBL

Filtered packets are routed via Network Address Translation to the Active Sink, per Figure 5.5. We used two separate filters: one for the Class A network and another for the two campus /19 subnets. We collected two sets of `tcpdump` traces for the networks: prefiltered traces with of packet headers, which we use in passive measurements (of periods during which the active responders were turned off), and filtered traces with complete payloads, which we use for active traffic analysis. The prefiltered traces for the Class A network are sampled at 1/10 packets to mitigate storage requirements.

LBL Sink: The LBL Sink monitors two sets of 10 contiguous /24 subnets. The first is for passive analysis; we merely listen but do not respond, and we do *not* filter the traffic. The second is for active analysis. We further divide it into two halves, 5 /24 subnets each, and apply filtering on these separately. After filtering, our system tunnels the traffic to the active responders, as shown in Figure 5.5. This tunnel is one-way—the responses are routed directly via the internal router. We use the same set of application protocol responders at LBL as in iSink, but they are invoked by Honeyd instead of iSink, because Honeyd is sufficient for the scale of traffic at LBL after filtering. We trace active response traffic at the Honeyd host, and unless stated otherwise this comes from one of the halves (*i.e.*, 5 /24 subnets).

Note that the LBL and UW campus have the same /8 prefix, which gives them much more locality than either has with the class A network.

Table 5.1 summarizes the datasets used in our study. At each network we collected passive `tcpdump` traces and filtered, active-response traces. On the two UW networks and the LBL network, we collected

Site	Networks (/size)	Datasets	Duration
iSink	UW-I (/19)	Active	Mar16–May14, 2004
		Passive	Mar11–May14, 2004
	UW-II (/19)	Active	Mar16–May14, 2004
		Passive	Mar11–May14, 2004
	Class A (/8)	Active	Mar12–Mar30, 2004
		Passive	Mar16–Mar30, 2004
LBL Sink	LBL-A (2 x 5 x /24)	Active	Mar12–May14, 2004
	LBL-P (10 x /24)	Passive	Apr 28–May 5, 2004

Table 5.1: Summary of Data Collection

two months' worth of data. Our provisional access to the class A enabled us to collect about two weeks of data.

The sites use two different mechanisms to forward packets to the active responder: tunneling, and Network Address Translation (NAT). The LBL site uses tunneling (encapsulation of IP datagrams inside UDP datagrams), which has the advantages that: (1) it is very straightforward to implement and (2) it does not require extensive state management at the forwarder. However, tunneling requires the receive end to (1) decapsulate traces before analysis, (2) handle fragmentation of full-MTU packets, and (3) allocate a dedicated tunnel port. NAT, on the other hand, does not have these three issues, but necessitates maintaining per-flow state at the forwarder, which can be significant in large networks. The stateless responder deployed at the UW site allows such state to be *ephemeral*, which makes the approach feasible. That is we only need to maintain a consistent flow ID for each outstanding incoming packet, so the corresponding flow record at the filter can be evicted as soon as it sees a response. Hence, the lifetime of flow records is on the order of milliseconds (RTT between the forwarder and active-sink) instead of seconds.

5.4 Passive Measurement of Background Radiation

This section presents a baseline of background radiation traffic on unused IP addresses *without actively responding to any packet*. It starts with a traffic breakdown by protocols and ports, and then takes a close look at one particular facet of the traffic: backscatter.

5.4.1 Traffic Composition

A likely first question about background radiation characteristics is "What is the type and volume of observed traffic?". We start to answer this question by looking at two snapshots of background radiation traffic shown in Table 5.2 which includes an 80 hour trace collected at UW Campus on a /19 network from May 1 to May 4, a one week trace at LBL collected on 10 contiguous /24 networks from April 28 to May 5, and finally a one-week trace at Class A with 1/10 sampling from March 11 to 18.

Protocol	UW-1		LBL-P		Class A	
	Rate	%	Rate	%	Rate	%
TCP	928	95.0%	664	56.5%	130	88.5%
ICMP	4.00	4.2%	488	39.6%	0.376	0.3%
UDP	0.156	0.8%	45.2	3.8%	16.5	11.3%

Table 5.2: Protocol breakdown by packet rate. The rate is computed as number of packets per destination IP address per day, *i.e.*, with network size and sampling rate normalized

Protocol	UW		LBL	
	#SrcIP	Percentage	#SrcIP	Percentage
TCP	759,324	87.9%	586,025	90.0%
ICMP	109,135	12.6%	64,120	9.8%
UDP	4,273	0.5%	4,360	0.7%

Table 5.3: Protocol breakdown by number of sources.

Clearly, TCP dominates more or less in all three networks. The relatively lower TCP rate at Class A is partly due to the artifact that the Class A trace was collected in March instead in May, when we see a few large worm/malware outbreaks (include the Sasser worm). Not shown in the table, about 99% of the observed TCP packets are `TCP/SYN`.

The large number of ICMP packets (of which more than 99.9% are `ICMP/echo-req`) we see at LBL form daily high volume spikes (Figure 5.6), which are the result of a small number of sources scanning every address in the observed networks. On the other hand we see a lot fewer ICMP packets at the Class A monitor which is probably because the Welchia worm, which probes with `ICMP/echo-req`, avoids the Class A network.

Finally, the surprising low rate of UDP packets observed at UW is largely due to the artifact that UW filters UDP port 1434 (the Slammer worm).

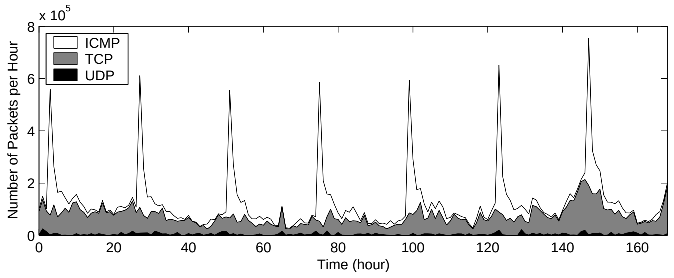


Figure 5.6: Number of background radiation packets per hour seen at LBL

In Figure 5.6, we can also see that `TCP/SYN` packets seen at LBL arrive at a relatively steady rate, (and this is the case for the other two networks as well) in contrast to daily ICMP spikes. A closer look at the breakdown of `TCP/SYN` packets by destination port numbers at LBL (Table 5.4) reveals that a small number of ports are the targets of a majority of `TCP/SYN` packets (the eight ports listed in the table account for more than 83% of the packets).

Table 5.3 shows the same traces from the perspective of the source of the traffic. Note that the rows are not mutually exclusive as one host may send both TCP and UDP packets. It is clear that TCP packets dominate in the population of source hosts we see. The distribution across ports of LBL traffic is shown in Table 5.4; as before, a small number of ports are dominant.

TCP Port	# Source IP (%)	# Packets (%)
445	43.4%	19.7%
80	28.7%	7.3%
135	19.1%	30.4%
1025	4.3%	5.8%
2745	3.2%	3.6%
139	3.2%	11.1%
3127	2.7%	3.2%
6129	2.2%	2.4%

Table 5.4: The Most Popular TCP Ports. Ports that are visited by the most number of source IPs, as in a one week passive trace at LBL. In total there are 12,037,064 packets from 651,126 distinct source IP addresses.

As `TCP/SYN` packets constitute a significant portion of the background radiation traffic observed on a passive network, the next obvious question is, *"What are the intentions of these connection requests?"*. We explore this question in Section 5.5 and 5.6.

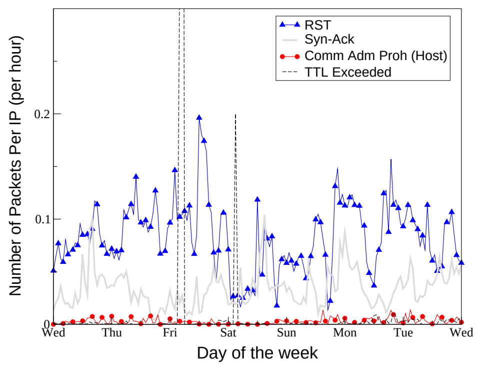
5.4.2 Analysis of Backscatter Activity

The term Backscatter is commonly used to refer to unsolicited traffic that is the result of responses to attacks spoofed with a network's IP address. We assume that packets observed at network telescopes with certain types of TCP flags, such as SYN-ACK and RST, or ICMP message types, such as ICMP Unreachable and TTL-exceeded, are backscatter traffic, because these packets are usually generated in response to other packets, but themselves do not solicit responses—so there is no incentive to send these packets intentionally to random addresses.

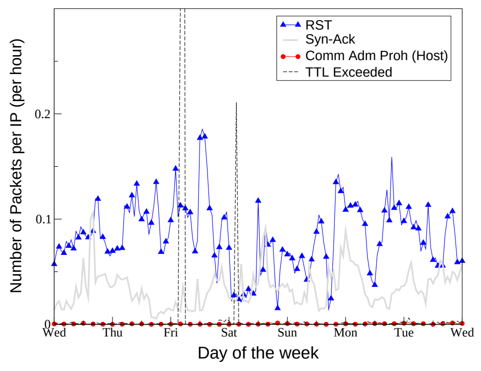
Figure 5.7 provides a time-series graph of the backscatter activity seen on the four networks. Not surprisingly, `TCP/RSTs` and `SYN-ACKs` account for the majority of the scans seen in all four networks. These would be the most common responses to a spoofed SYN-flood (Denial of Service) attack. The figures for the two UW and the Class A networks span the same two weeks. The backscatter in the two UW networks looks highly similar both in terms of volume and variability. This can be observed both in the TCP RSTs/SYN-ACKs and the two surges in ICMP TTL-Exceeded shown in Figures 5.7(a) and (b), and makes sense if the spoofed traffic which is eliciting the backscatter is uniformly distributed across the UW addresses. The only difference between the networks is that UW I tends to receive more "Communication administratively prohibited" ICMP messages than UW II. We do yet have an explanation why. While we see some common spikes in the `SYN-ACKS` at the Class A and UW networks, there seem to be significant differences in the `RSTs`. Another notable difference is that the Class A network attracts much more backscatter in other categories, as shown in Figure 5.8.

The LBL graph shown in Figure 5.7(c) belongs to a different week and displays a quite different pattern than that of UW. We note that the backscatter in the UW networks for the same week (not shown here) shows a very similar pattern as at LBL for the dominant traffic types (TCP `RSTs`/`SYN-ACKs` and ICMP `TTL-Exceeded`). This is not surprising, because the two UW networks and the LBL network belong to the same /8 network. On the other hand, the LBL network seems to receive far fewer scans in the other categories.

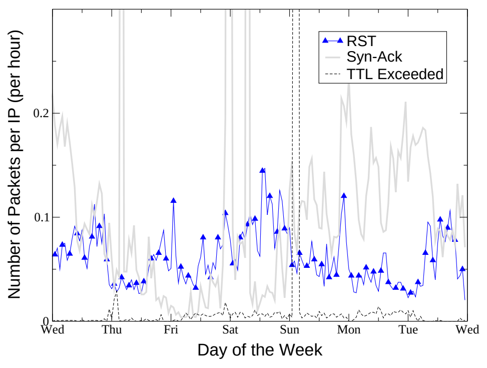
A significant portion of ICMP `host-unreach` messages we see at Class A are responses to UDP packets with spoofed source addresses from port 53 to port 1026. We first thought we were seeing backscatters of DNS poisoning attempts, but then we found that we are also seeing the UDP packets in other networks as well. Examining these packets reveals that they are not DNS packets, but rather Windows Messenger Pop-Up spams, as discussed in the next section.



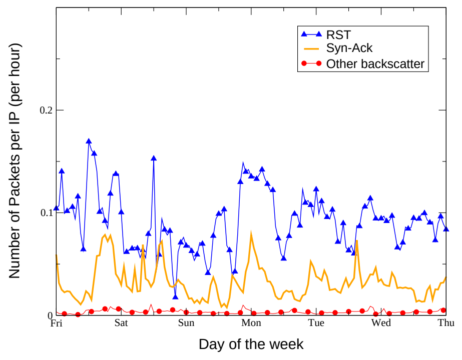
(a) UW I



(b) UW II



(c) LBL



(d) Class A (see Figure 5.8 for further breakdown of "other")

Figure 5.7: Time series of weekly backscatter in the four networks.

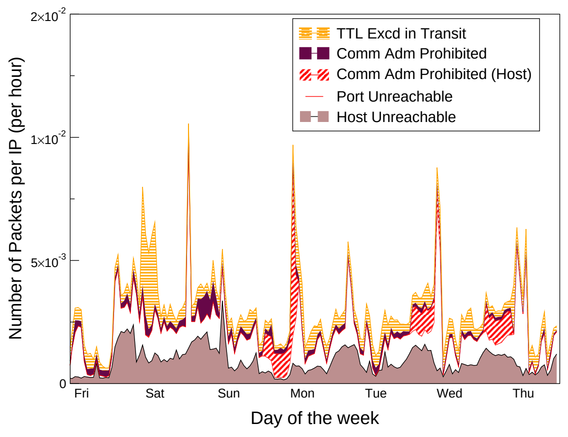


Figure 5.8: The components of backscatter at the class A network besides the dominant RST, SYN-ACKs.

5.5 Activities in Background Radiation

In this section we first divide the traffic by ports and present a tour of dominant activities on the popular ports. Then we add the temporal element to our analysis to see how the volume of activities vary over time.

5.5.1 Details per Port

We rank activities' popularity mostly by number of source IPs, rather than by packet or byte volume, for the following reasons. First, our filtering algorithm is biased against sources that try to reach many destinations, thus affects packet/byte volumes unevenly for different activities. The number of source IPs, however, should largely remain unaffected by filtering, assuming a symmetry among destinations, i.e. when a source contacts a number of destinations, the response (or lack of one) from one host will not affect the semantics of traffic to other hosts, though the rate of traffic can be affected. Also, number of source IPs reflects popularity of the activity across the Internet—an activity with a huge number of sources is likely to be prominent on the whole Internet. Finally, while a single-source activities might be merely a result of an eccentric host, a multi-source activity is more likely to be intentional.

When a source host contacts a port, it is common that it sends one or more probes before revealing its real intention, sometimes in its second or third connection to the destination host. A probe can be an empty connection, *i.e.* the source opens and closes the connection without sending a byte, or some short request, *e.g.*, an HTTP `"GET /"`. Since we are more interested in the intention of sources, we choose to look at the activities at a per-session (source-destination pair) granularity rather than a per-connection granularity. Otherwise one might reach the conclusion that the probes are the dominant elements. We consider all connections between a source-destination pair on the given destination port collectively and suppress repetitions. This approach usually gives us a clear picture of activity on each port.

Below we examine the activities on popular destination ports, and for each port we present the dominant activities. For convenience of presentation, we introduce abbreviations for activity descriptions, as shown in Table 5.5. We pick an arbitrary day, March 29, 2004, to compare the distribution of activities seen at different networks, LBL, UW (I,II), and the Class A network. We consider the two UW networks as a single network to eliminate possible bias that might occur due to a single filter.

Port/Abbrev.	Activity
80/Get	`"GET /"`

80/GetSrch	`"GET /"`
	`"SEARCH /"`

80/SrchAAA	`"GET /"`
	`"SEARCH /"`
	`"SEARCH /AAA..."`

80/Srch64K	`"SEARCH /\x90\x02\xb1\x02\xb1..."` (65536 byte URI)
135/Bind1	`RPC bind: 000001a0-0000-0000-c000-000000000046`
135/RPC170	`Unknown RPC request (170 bytes)`
135/Bla	`RPC exploit: Blaster`
135/Wel	`RPC exploit: Welchia`
135/RPC-X1	`RPC exploit: (1624 bytes)`

135/EP24-X2	`(Empty connection on port 135/tcp)`
	`RPC request (24 bytes)`
	`RPC exploit: (2904 bytes)`

445/Nego	`(CIFS session negotiation only)`
445/Locator	`"\\<ip>\IPC$ \locator" => RPC exploit: 1896 bytes`

445/Samr-exe	`"\\<dst-IP>\IPC$ \samr"`
	`"\\<dst-IP>\IPC$ \srvsvc => CREATE FILE: "[...].exe"`

445/Samr	`"\\<dst-IP>\IPC$ \samr"`
445/Srvsvc	`"\\<dst-IP>\IPC$ \srvsvc"`
445/Epmapper	`"\\<dst-IP>\IPC$ \epmapper"`

Table 5.5: Abbreviations for Popular Activities. Each line reflects a separate connection.

The background radiation traffic is highly concentrated on a small number of popular ports. For example, on March 29 we saw 32,072 distinct source IPs at LBL,[3] and only 0.5% of the source hosts contacted a port not among the "popular" ports discussed below. Thus by looking at the most popular ports, we cover much of the background radiation activity.

Note that looking at the ports alone does not allow us to distinguish the background radiation traffic, because many of the popular ports, *e.g.,* 80/tcp (HTTP), 135/tcp (DCE/RPC) and 445/tcp (SMB), are also heavily used by the normal traffic. On the other hand, once we look at the background radiation traffic at application semantic level, it has a very distinctive modal distribution. For example, the activities on port 135 are predominantly targeted on two particular DCE/RPC interfaces, and almost all buffer-overrun exploits are focused on one interface. It is worth noting that the activity composition may change dramatically over time, especially when new vulnerabilities/worms appear, *e.g.,* the dominant activity on port 445 is no longer "Locator" after the rise of the Sasser worm. However, we believe the modal pattern will last as long as the background radiation traffic remains highly automated.

[3]Here we ignore the effect of source IP spoofing, since our responder was able to establish TCP connections with most of the source hosts.

Activity	LBL	UW	Class A
Get	5.1%	2.9%	4.6%
GetSrch	5.2%	93.2%	93.4%
SrchAAA	84.2%	0.0%	0.0%
Srch64K	0.9%	1.1%	0.0%
CodeRed	0.6%	0.4%	0.5%
Nimda	0.2%	0.1%	0.2%
Other	3.8%	2.3%	1.3%

Table 5.6: Port 80 Activities (March 29, 2004) Note that to reduce trace size the active responders at UW and Class A do not respond to `"SEARCH /"` to avoid getting the large `SrchAAA` requests.

TCP Port 80 (HTTP) and HTTP Proxy Ports: Most activities we see on port 80 (Table 5.6) are targeted against the Microsoft IIS server. In most cases, imitating the response of a typical IIS server enables us to attract follow-up connections from the source.

The dominant activity on port 80 is a WebDAV buffer-overrun exploit [126] (denoted as SrchAAA). The exploit always makes two probes: `"GET /"` and `"SEARCH /"`, each in its own connection, before sending a `"SEARCH"` request with a long URI (in many cases 33,208 bytes, but the length can vary) starting with `"/AAAA..."` to overrun the buffer. Unlike exploits we see on many other ports, this exploit shows a lot of payload diversity—the URIs can be different from each other by hundreds of bytes. More interestingly, the URIs are composed solely of lower-case letters except for a few dozens of Unicode characters near the beginning. The Unicode section turns out to be a short decoder which translates the remaining characters in the URI to executable x86 code. Besides this exploit, we also see other WebDAV exploits, *e.g.*, one popular exploit (Srch64K) from Agobot carries a fixed 65,536 byte URI.

Old IIS worms, Nimda and CodeRed II, remain visible in the datasets. The CodeRed II worm is almost the same as the original CodeRed II, except shift of a space and the change of expiration date to year 0x8888. We also often see a `"OPTIONS /"` followed by a `"PROPFIND"` request. As both requests are short, they look like probes. We have not been able to elicit further requests from the sources and do not yet fully comprehend the intention behind such probes. We suspect that they might be scanners trying to obtain a listing of list of scriptable files by sending “translate: f” in the header of the HTTP request [105].

An interesting component of background radiation observed across all networks on the HTTP proxy ports: 81/1080/3128/8000/8080/8888,[4] as well as on port 80, is source hosts using open-proxies to send probes to `tickerbar.net`. A typical request is shown in Figure 5.9. These requests are from sources

[4]Though some of these ports are not officially assigned to HTTP, the traffic we received almost contained only HTTP requests.

abusing a “get rich quick” money scheme from greenhorse.com–a web site pays users money for running tickerbar while they surf the net. By using open-proxies, these sources can potentially appear to be running hundreds of nodes. The Greenhorse website seems to have since been inactivated.

```
GET http://dc.tickerbar.net/tld/pxy.m?nc=262213531 HTTP/1.0
   Host: dc.tickerbar.net
   Connection: Close
```

Figure 5.9: Typical HTTP request of a tickerbar host

TCP Port 135/1025 (DCE/RPC): Port 135 is the Endpoint Mapper port on Windows systems [25] and one of the entry points to exploit the infamous Microsoft Windows DCOM RPC service buffer overrun vulnerability [124]. This vulnerability is exploited by the Blaster worm and the Welchia worm among others.

Figure 5.10 shows the dominant activities on the port. The Blaster worm was seen on all three networks, but strangely we only saw the Welchia worm at LBL. There were also a number of empty connections without follow-ups and a few types of probes (*e.g.,* 135/RPC170) we do not understand well. Comparing the activity distribution across three networks, the difference is striking and unlike what we see on other ports. This may be due to 1) lack of a single dominant activity and 2) that certain scanning and exploits might be targeted or localized.

On port 1025, which is open on a normal Windows XP host, we see a similar set of exploits. Further, DCE/RPC exploits are also seen on CIFS named pipes on port 139 and 445. We present a closer look at RPC exploit in Section 5.5.2.

TCP Port 139/445 (CIFS): Port 139 is the NetBIOS Session Service port and is usually used on Windows systems for CIFS (Common Internet File System) [19] over NetBIOS. Port 445 is for CIFS over TCP and is also known as Microsoft-DS. When used for CIFS sessions, the two ports are almost identical except that NetBIOS requires an extra step of session setup. Sources simultaneously connecting to both ports prefer port 445 and abandon the port 139 connection. Thus we frequently see empty port 139 connections.

As many Windows services run on top of CIFS there are a great variety of exploits we see on these two ports. Figure 5.2 shows a snapshot of exploits we see on port 445 at the Class A network. There are basically two kinds of activities: 1) buffer-overrun RPC exploits through named pipes, *e.g.* the Locator pipe [125] or the Epmapper pipe (connected to the endpoint mapper service); and 2) access control bypassing followed by attempts to upload executable files to the target host, *e.g.* as in exploit 445/Samr-exe.

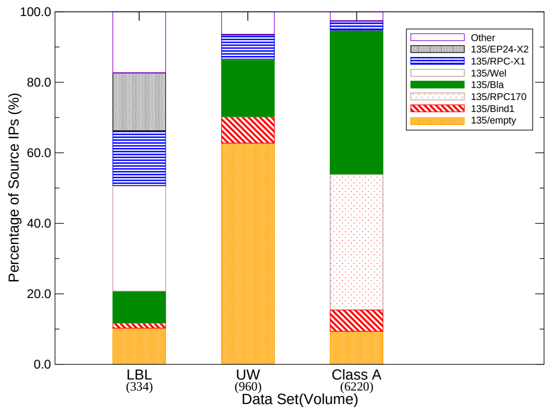


Figure 5.10: Port 135 activities on March 29

Activity	LBL	UW	Class A
445/empty	2.4%	1.3%	0.9%
445/Nego	3.3%	2.4%	3.7%
445/Locator	72.7%	89.4%	89.3%
445/Samr-exe	11.6%	1.8%	1.1%
445/Samr	2.7%	0.8%	0.6%
445/Srvsvc	1.1%	0.4%	0.8%
445/Epmapper	0.8%	0.3%	0.0%
Other	5.4%	3.7%	3.5%

Table 5.7: Port 445 activities

As shown in Table 5.7, the Locator pipe exploit dominates port 445 activities at all four networks. Besides that, some sources did not go beyond the session negotiation step—the first step in a CIFS session. We also see exploits that first connect to the SAMR (Session Account Manager) pipe, then connect to the SRVSVC pipe and attempt to create an executable file with names such as `msmsgri.exe`(W32 Randex.D) [99] and `Microsoft.exe` [2]. Finally, by connecting to the Epmapper pipe the sources are exploiting the same vulnerability as on port 135/1025—note that this activity is not seen in the Class A network.

On port 139, 75% to 89% of source hosts either merely initiate empty connections or do not go beyond the NetBIOS session setup stage, and then migrate to port 445; The dominant activity that we accurately identify are attempts to create files on startup folders after connecting to the SRVSVC pipe `Xi.exe`(W32-Xibo) [131].Unlike port 445, we see few hosts attempting to exploit the buffer overflows on the Locator or Epmapper pipe. We also see Agobot variants that connect to the SAMR pipe and drop executables.

TCP Port 6129 (Dameware): Dameware Remote Control, an administration tool for Windows systems, listens on port 6129. Dameware has a buffer overrun vulnerability in its early versions [123]. The Dameware exploits we see are similar to those of published exploit programs but do not have exactly the same payload. To launch an exploit, the source host will first send a 40 byte message to probe operating system version and then ship the exploit payload, which is almost always 5,096 bytes long.

On March 29, 2004, 62% of the source hosts that connect to port 6129 at LBL[5] close the connections without sending a byte; another 26% abandoned the connections after sending the probe message; and we see exploit messages from the remaining 12% (the number is over 30% on Apr 29). It would be reasonable to question if the large number of abandoned connections suggest that the sources did not like our responders. However, we also find source hosts that would first connect with an empty connection and later came back to send an exploit. Port 6129 is associated with the Agobot that connects to a variety of ports (see Section 5.6.1), and possibilities are that the bots may connect to a number of ports simultaneously and decide to exploit the port that they receive a response from first.

TCP Port 3127/2745/4751 (Virus Backdoors): Port 3127 and 2745/4751 are known to be the backdoor ports of the MyDoom and Beagle viruses, respectively. On most port 3127 connections, we see a fixed 5-byte header followed by one or more Windows executable files uploads. The files are marked by `"MZ"` as the first two bytes and contain the string `"This program cannot be run in DOS mode"` near the head of the file. Running several captured executable files in a closed environment reveals that the programs scan TCP ports 3127, 135, and 445.

On port 2745, the dominant payload we see at LBL and UW is the following FTP URL, which comes after exchanging of one or two short binary messages.

```
"ftp://bla:bla@<src-IP>:<port>/bot.exe\0"
```

On the Class A network, however, we do not see a lot of port 2745 activities. Interestingly, we see several source hosts that attempt to upload Windows executables. We also see many hosts that close the connection after exchange of an initial message.

On port 4751, in some cases we see binary upload after echoing a header, similar to what happens on port 3721, but in most cases we receive a cryptic 24-byte message, and are unable to elicit further response by echoing.

[5]Due to an iSink responder problem we do not have data for the UW and Class A network.

```
20:27:43.866952 172.147.151.249.domain > 128.3.x.x.domain: [udp sum ok]
    258 [b2&3=0x7] [16323a] [53638q] [9748n] [259au]
    Type26904 (Class 13568)? [|domain] (ttl 115, id 12429, len 58)
0x0000   (...)
0x0010   xxxx xxxx 0035 0035 0026 xxxx 0102 0007        ................
0x0020   d186 3fc3 2614 0103 d862 6918 3500 d54c        ..?.&....bi.5..L
0x0030   8862 3500 cb1f ee02 3500                       .b5.....5.
```

Figure 5.11: Unknown packets on UDP port 53 (DNS port)

TCP Port 1981/4444/9996: (Exploit Follow-Ups): While worms such as CodeRed and Slammer are contained completely within the buffer-overrun payload, several of the other worms such as Blaster and Sasser infect victim hosts in two steps. First, the buffer-overrun payload carries only a piece of "shell code" that will listen on a particular port to accept further commands. Second, the source then instructs the shell code to download and execute a program from a remote host. For example, on port 4444, the follow-up port for the Blaster worm, we often see:

```
tftp -i <src-IP> GET msblast.exe
start msblast.exe
msblast.exe
```

Similarly, on port 1981 (Agobots) and 9996 (Sasser) we see sequences of shell commands to download and execute a `bot.exe`. In contrast, there is a different kind of shell code called "reverse shell" which does not listen on any particular port, but instead connects back to the source host ("phone home"). The port on the source host can be randomly chosen and is embedded in the shell code sent to the victim. The Welchia worm uses a reverse shell (though its random port selection is flawed). This makes it much harder to capture the contents of follow-up connections, because 1) we will have to understand the shell code to find out the "phone-home" port; and 2) initiating connections from our honeypots violates the policy of the hosting networks.

UDP Port 53: We expected to see a lot of DNS requests, but instead, find sources sending non-DNS (or malformed) packets as shown in Figure 5.11.

We do not know what these packets are. These requests dominate UDP packets observed in the LBL and UW (I,II) networks.

Table 5.8 provides a summary of the DNS activity observed in the Class A network during a 24 hour trace showing a more diverse activity. Much like the UW and LBL networks, sources sending malformed DNS requests dominate. However, in terms of packet counts other queries are substantial. We suspect

Type	Num packets	Num sources
Malformed packets	5755	3616
Standard (A) queries	10139	150
Standard query (SOA)	4059	95
Standard query (PTR)	1281	27
DNS Standard query SRV packets	785	20
DNS Standard query AAAA packets	55	16
DNS Standard unused packets	739	3
DNS Standard unknown packets	1485	3

Table 5.8: Summary of DNS activity seen in the Class A (24 hours)

these are possibly due to misconfigured DNS server IP addresses on hosts. These queries are sent to various destination IP addresses and originate from various networks. Hence it seems unlikely that these are a result of stale DNS entries.

The biggest contributor in terms of volume are standard A queries that resolve IP address for domain names. The SOA packets are “Start of Authority” packets used to register domain authorities. We observed 45 sources (out of total 95) registering different domain authorities in BGC.net. Other queries include PTR queries (used for reverse DNS lookups), SRV records (used to specify locations of services) and AAAA queries (IPv6 name resolution).

UDP Port 137: The activities are dominated by NetBIOS standard name queries (probes).

UDP Port 1026, 1027 (Windows Messenger Pop-Up Spam): These appear as UDP packets with source port 53 and destination port 1026 (or 1027). While this port combination typically connotes a DNS reply, examination of packet contents reveals that they are in fact DCE/RPC requests that exploit a weakness in the Windows Messenger API to deliver spam messages to unpatched Windows desktops [129]. Figure 5.12 shows a trace of a typical packet. The source IP addresses of these packets are often spoofed, as suggested by the observed `ICMP host-unreach` backscatter of these attacks in the Class A. The choice of source port 53 is most likely to evade firewalls.

UDP Port 1434: The Slammer worm is still alive and is the only background radiation we see on port 1434.

TCP Port 1433: We have not yet built a detailed responder for MS-SQL. It appears that most source hosts are trying to log in with blank passwords.

```
05:23:16.964060 13.183.182.178.domain > xxx.xxx.xxx.xxx.1026:  1024 op5
    [4097q] 68/68/68 (Class 0) Type0[|domain] (DF)
...
0x0010   .... .... .... .... .... .... 0400 a880        ................
0x0020   1001 000a 000a 000a 0000 0000 0000 0000        ................
0x0030   0000 0000 f891 7b5a 00ff d011 a9b2 00c0        ......{Z........
0x0040   4fb6 e6fc 4ba6 e851 f713 8030 a761 c319        O...K..Q...0.a..
0x0050   13f0 e28c 0000 0000 0100 0000 0000 0000        ................
0x0060   0000 ffff ffff 6400 0000 0000 0c00 0000        ......d.........
0x0070   0000 0000 0c00 0000 5265 616c 2057 6f6d        ........Real.Wom
0x0080   656e 0000 0400 0000 0000 0000 0400 0000        en..............
0x0090   596f 7500 3000 0000 0000 0000 3000 0000        You.0.......0...
0x00a0   5741 4e54 2053 4558 3f0d 0a0d 0a46 494e        WANT.SEX?....FIN
0x00b0   4420 5553 2041 543a 0d0a 0d0a 0977 7777        D.US.AT:.....www
0x00c0   2exx xxxx xxxx xxxx xx2e 4249 5a0d 0a00        .********.BIZ...
```

Figure 5.12: Observed Windows Messenger Pop-Up Spam packets.

TCP Port 5000: We do not know enough about this port. The port is reserved for Universal Plug-and-Play on Windows Systems, but almost none of requests we see are valid HTTP requests. However, most requests contain a number of consecutive 0x90's (`NOP`) and thus look like buffer-overrun exploits.

All the ports we examine above exhibit a modal distribution at the application semantic level, *i.e.,* they all contain one or a few dominant elements. The only exception is the DCE/RPC ports, on which we see some diversity, but in some sense, the various exploits on DCE/RPC ports have a single dominant element on a higher level—they target the same vulnerability. As the dominant elements are quite different from what we see in the normal traffic, this suggests that we will be able to capture the majority of background radiation traffic with a sound classification scheme at the application semantic level.

5.5.2 Temporal Distribution of Activities

We examine two cases of temporal activity. First, we look at the exploits with the largest source population and consider the population variation over time. Second, we look at the exploits targeted at a particular vulnerability and consider how these exploits evolve and diversify over time. We focus on the LBL network for this analysis.

The Dominant Exploits

Figure 5.13 shows how the numbers of source hosts vary over the course of 18 days for the four exploits with largest source population.

The source volumes for the SrchAAA and Locator exploits are relatively stable and close to each other over time. This is not surprising because these exploits are likely coming from the same worm, as we see in Section 5.6.2.

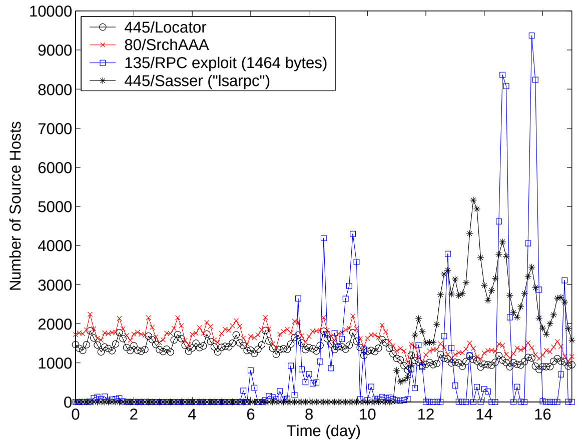


Figure 5.13: The Big Exploits (Apr 20 to May 7, 2004), as observed on 5 /C networks at LBL. The source hosts are counted every three hours.

The other two exploits, Exploit1464 and Sasser, show much a wider range of source volume dynamics. This is especially true for Exploit1464, which temporarily retreated to a much smaller scale around April 30th.

All four exploits demonstrate a strong diurnal pattern, with obvious peaks at local time noon. We do not have good explanations for this pattern. For SrchAAA, Locator, and the Sasser exploits, the peak might be due to hosts being turned on at daytime and doing local-biased search. However, for Exploit1464, the steep narrow peaks lead us to believe they could be caused by the scanning mechanism itself.

Overall, we can see that there are two common types of temporal patterns in background radiation at the granularity of every few hours: the source population of some radiation activities (such as SrchAAA) remains largely constant, likely a result of continual and independent random scanning by the source hosts; some other radiation activities (such as Exploit1464) exhibit large variations, suggesting synchronized or centrally controlled scanning.

We also observe variation of source population at larger time granularity—in terms of days or weeks. Such variation can be a result of a new vulnerability and therefore new types of exploits, as shown by the Sasser exploit, which starts to appear around the April 30, likely coming from the Sasser worm. On the other hand, the source population of existing radiation activities may decrease as new types of malware emerge and take over some of the hosts, as the SrchAAA and Locator population does upon the Sasser worm outbreak.

DCE/RPC Exploits

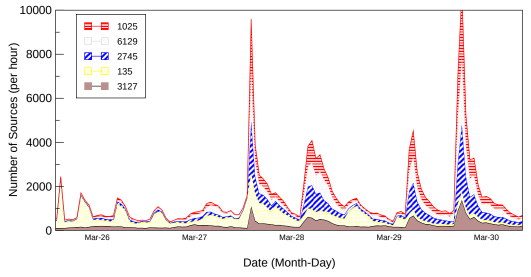
DCE/RPC exploits that target the Microsoft DCOM RPC vulnerability [124] present an interesting case of a single well-known vulnerability being used and reused by various worms and/or autorooters. This vulnerability is particularly attractive because it exists on every unpatched Windows 2000/XP system, in contrast to, *e.g.* vulnerabilities that exist only on IIS or SQL servers.

We have seen quite a few different exploit payloads in this data. There are at least 11 different payload lengths. This does not appear to be result of intentional polymorphism, for two reasons: 1) from almost every single source IP we see only one payload length; and 2) it would be easy to vary the length of payload by simply inserting `NOP`'s if the adversary wanted to incorporate some polymorphism. Thus we believe that the diversity of payloads is not due to deliberate polymorphism but due to different code bases. While the payloads themselves might not be very interesting, since the diversity is likely due to the various "shell code" they carry, the diversity offers us an opportunity to look at the rising and ebbing of different exploit programs.

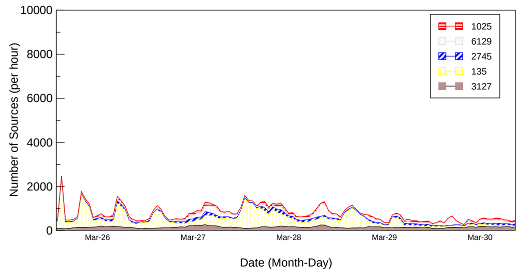
Without a robust way to cluster payloads by contents (payloads of same length sometimes differ on tens to hundreds of bytes and the differences are not merely shifting of contents and paddings), we choose to cluster the exploits by lengths and the ports on which they appear, including port 135/1025 and the Epmapper pipes on port 445/139. Under this scheme, we see more than 30 different exploit types. We select nine of the popular exploits and consider how the number of source IP addresses for each exploit varies over time during April 2004. The exploits have four different payload lengths: 1448, 1464, 2972, and 2904, and are seen on port 135, 1025, and 445.

We observe strong temporal correlation among exploits of the same length for lengths 2792 and 2904, while this is not the case for lengths 1448 and 1464. The four exploits also show some correlation in terms of activity to port 135 and to port 445, which is due to the same source probing both ports. We also find that even for multiple sources, activity for particular port/length pairs tends to come in bursty spikes, suggesting that certain malware performs synchronized scanning among the sources.

The temporal correlation among some types of DCE/RPC exploits allows us to guess which exploits are coming from the same malware. It shows that some malwares probably send different exploits to attack the same vulnerability, but it is not clear why they do so (the variation is too little for evading signatures).



(a) Agobot Sources: UW I



(b) Agobot Sources: UW II

Figure 5.14: Time series of activity on Agobot ports in the two UW /19 networks (on adjacent Class B networks)

5.6 Characteristics of Sources

This section examines the background radiation activities in terms of source hosts. We associate various activities coming from the same source IP to construct an “activity vector” for each source IP, which we then examine in three dimensions: 1) across ports, 2) across destination networks, and 3) over time.

One caveat here is the possibility of IP spoofing. Yet we find that we are able to establish two-way conversation with the majority of source hosts in background radiation, suggesting that most sources are not spoofing their IP addresses.

There is another caveat about identifying hosts with IP addresses: due to DHCP, hosts might be assigned different addresses over time. A study [71] concluded that “IP addresses are not an accurate measure of the spread of a worm on timescales longer than 24 hours”. However, without a better notion to identify hosts, we still use IP addresses to identify hosts, while keeping this caveat in mind.

5.6.1 Across Ports

Associating activities across ports sometimes gives us a significantly better picture of a source’s goals. This especially helps with analyzing puzzling activities, because it puts behavior on individual ports in the context of collective activity. For example, simply looking at an RPC exploit may not readily reveal the worm or autorooter that sends it, but once we see a follow-up to port 4444 with `"tftp msblast.exe"`, we know that the earlier exploit comes from Blaster.

Table 5.9 provides a summary of the top multi-port scanning episodes seen in the four networks. The most common is sources that scan both 139 and 445. Many viruses that exploit NetBIOS/SMB (CIFS) exhibit this behavior since for propagation the ports can be used interchangeably. This next most common

Name	Ports	Description	Number of Sources (Rank)			
			LBL	UW I	UW II	Class A
NB-1	139,445	Xi.exe (W32-Xibo), msmsgri32.exe (Randex.D), Antivirus32.exe (SDBot.JW)	4,310 (1)	4,300 (1)	4,313 (1)	7,408 (1)
NB-EP1	135,139,445	EP-2704, mdms.exe(Agobot)	1,187 (2)	1,028 (2)	1,046 (2)	537 (4)
NB-EP2	135,139, 445,5000	EP-2792, EP-2904	780 (3)	678 (3)	721 (3)	15
Agobot-1	1025,1981, 2745,6129	Agobot variant-I	16	452 (4)	68 (10)	0
Agobot-2	1025,2745,6129	Agobot variant-II	1	437 (5)	3	0
Agobot-3	80,139,1025, 2745,6129	Agobot variant-III	0	415 (6)	0	0

Table 5.9: 24 hours of multi-port source activity at the four sites

multiport source behavior exploits the Microsoft DCE/RPC vulnerability [124] both via port 135 and by connecting to the Epmapper pipe through port 139 and port 445 during the same episode. These are likely variants of Welchia. Most port 5000 connections are empty and the rest small portion of them look like buffer overrun exploits. We also find Agobot variants that occasionally target these services. They connect to the SAMR pipe through CIFS to obtain registry information, following the sequence described in Figure 5.3 and drop the file `mdms.exe` into one of the startup folders. The least common profiles are used exclusively by Agobot variants (I, II and III).

We can examine the spatial variance of multi-port profiles by comparing the data collected at each network during the same 24 hour period. We order the profiles based on the ranks of all multi-port profiles collected at the UW I network, which showed the greatest affinity to Agobot. The table reveals several notable observations. First, the top two exploits are extensively observed across all four networks and their rankings are consistent in the Class B networks—a *spatial invariant*. Second, although the LBL network is much smaller, it observes the same number of sources as the other two UW networks for the top three exploits. This is probably due to the fact that the networks belong to the same /8, and suggests that these multi-port sources often sweep the address space. Third, UW I receives many more Agobot scans than the other networks, though we don't know why.

We explore Agobot scans in greater detail in Figure 5.14. The figures show the volume of unique sources per hour on several of the Agobot ports during a five-day period in March at the two adjacent UW networks. The graph at UW I shows four visible spikes, indicating an Agobot attack. While UW II

Activity	LBL	UW	Class A	LBL ∩ UW	LBL ∩ Class A
All	31K	276K	582K	15K	6.5K
Srch+Loc	76%	85%	57%	75%	91%
Samr-exe	1,601	2,111	2,012	1,634	116
Witty	72	241	162	61	18

Table 5.10: Traffic from sources seen across networks: intersections vs. individual networks

background radiation seems to otherwise closely follow UW I, these Agobot spikes are peculiarly absent. These graphs also provide a temporal perspective on the growth of Agobot, with a striking daily spike-followed-by-decay pattern, presumably as new machines are cleaned up over the course of the day.

We see little Agobot on the Class A network. This likely reflects Agobot's "maturity" as malware. It has gone through iterative enhancements and likely has (essentially) "evolved" to have been programmed to avoid unused class A networks; or perhaps it has become equipped with list of target networks, or the scanning is being consciously focused by a human operator via IRC control channels.

5.6.2 Sources Seen Across Networks

We now consider sets of source hosts seen on multiple networks at approximately the same time. We analyze source IPs seen across networks on an arbitrarily chosen day (March 29 GMT), characterizing them in terms of: 1) How many such source hosts are there? 2) Do they send the same traffic to different networks? 3) What does the activity distribution look like? and 4) how does it compare to the distribution on individual networks?

As shown in Table 5.10, source IPs seen at LBL and UW have a surprisingly large intersection set—almost half the source IPs seen at LBL are also seen at UW. In contrast, the intersection of LBL and the Class A is much smaller, even though we are seeing many more source IPs at the Class A than at UW.[6] This contrast may be due to some sources avoiding the Class A networks, and also the proximity of LBL and UW in the IP address space.

The next evaluation is to confirm that a given source indeed sends the same traffic to the different networks. We extract an activity vector for each source IP on the LBL and UW networks and compare, finding that indeed this is the case, with one peculiarity: while several thousand SrchAAA and Locator sources are common, we also find nearly two thousand Locator-only sources at one network that are SrchAAA-only

[6]Since we only see ICMP Unreachable backscatter only on the Class A network, and these constitute a significant number of source IPs, here we exclude them from the comparison to avoid skewing the activity distribution.

	Mar 29	Mar 30	Apr 29	1-Day ∩	1-Month ∩
All	31K	30K	62K	1,513	680
Srch+Loc	76%	83%	42%	68%	85%
Witty	72	64	0	15	0
Blaster	30	31	24	8	7

Table 5.11: Traffic from sources seen over time: intersections vs. individual periods

sources at the other. This turns out to be due to the interaction between source-destination filtering and the scanning mechanism of the SrchAAA/Locator sources. These sources choose, apparently randomly, to send either SrchAAA or Locator to a given destination, but not both.

Finally, what does the activity profile of a given source tell us about how likely we are to see it elsewhere? As shown in Table 5.10, sources exhibiting the dominant activity profile—SrchAAA and Locator—are often seen at multiple locations in the network. On the other hand, Samr-exe and Witty present an interesting case. The Samr-exe sources we see in the intersection of LBL and UW are more than what we find at LBL alone! (1,634 vs 1,601) This seeming inconsistency is caused by a number of source hosts not completing the exploit when contacting LBL, and thus not being identified there, but doing so at UW. In addition, the Samr-exe population seen at UW is merely 2,061 (0.7%), so we see a surprisingly large overlap for it between LBL and UW. On the other hand, LBL and the Class A have only 116 Samr-exe sources in common, out of more than 2,000 seen on the Class A, suggesting that Samr-exe sources scan with a local bias.

5.6.3 Sources Seen Over Time

To characterize sources seen at the same network over time, we analyze activity seen at LBL on three days: March 29, March 30, and April 29. This gives us comparisons for adjacent days and one month apart, respectively. Table 5.11 characterizes the variation. We see that the intersection of source hosts—even in the case of only one day apart—is much smaller than the intersection across networks. While this is partly because the UW network is larger than LBL, looking at the set of sources seen on another LBL network of the same size on March 29 we find more than 5,000 hosts in common. This confirms that we tend to see a larger intersection of source IPs across networks than over time. One effect we have not controlled for here, however, is DHCP artifacts: a host might be assigned different addresses on different days. We also note that the intersection size does not decrease very much further over one month's time, suggesting that if a host does not have the DHCP artifact, then it tends to stay in the intersection. The initial steep decaying of source IP sets also suggests that it will be easier to track a (malicious) host across space than across time.

The number of source hosts seen over time also varies by activity. For example, Witty did not persist over a month (nor could it, as it was a rare instance of a worm that deliberately damages its host); Blaster's grip on hosts is quite tenacious; and the SrchAAA/Locator sources fall in between.

5.7 Summary

Previous studies of Internet traffic have identified a number of now well-established properties: diurnal cycles in volume; variability in mix across sites and over time; bursty arrivals; the ubiquity of heavy-tailed distributions. Over the past several years, however, an important new dimension of Internet traffic has emerged, and it has done so without any systematic observation or characterization. The gross features of this new breed of traffic are that it is complex in structure, highly automated, frequently malicious, and mutates constantly. Each of these characteristics motivates the need for a deeper understanding of this "unwanted" traffic.

We have presented an initial study of the broad characteristics of Internet *background radiation*. Our evaluation is based on traffic measurements from four large, unused subnets within the IPv4 address space. We developed filtering techniques and active responders to use in our monitoring, analyzing both the characteristics of completely unsolicited traffic (passive analysis) and the details of traffic elicited by our active responses (activities analysis).

Passive analysis demonstrates both the prevalence and variability of background radiation. Evaluation of destination ports reveals that the vast majority of traffic targets services with frequently-exploited vulnerabilities. Analysis of backscatter traffic shows the overall dominance of TCP SYN-ACK/RST packets, but otherwise we do not find a great deal of consistency across the monitored subnets.

Our activities analysis focused on the most popular services targeted by background radiation, finding a rich variegation. Activities across all of the monitored services include new worms released during our study, vestiges of old worms such as Code Red and Nimda, the frequent presence of "autorooter" scans (similar to worms, but without self-propagation), and a noticeable number of connections that are simply empty even when given an opportunity to send data. As with the passive analysis, we find significant diversity across the subnets we monitored, and also over time.

We also examined background radiation from the perspective of source host behavior. Considering source activities across ports reveals consistent behavior in each of the measurement sites for the most prevalent multi-port scan type (scans to both ports 139 and 445). Furthermore, there was an appreciable intersection of sources across measurement sites. This can be explained by the random scanning behavior of worms like Welchia. However, there was a much smaller set of sources common to all measurement sites when they are considered over time.

Perhaps the most striking result of our analysis is the extreme dynamism in many aspects of background radiation. Unlike benign traffic, which only shows major shifts in constituency when new applications become popular (which happens on fairly lengthy time scales), the mix of background radiation sometimes changes on a nearly-daily basis. This dynamism results in a potpourri of connection-level behavior, packet payloads, and activity sessions seen in different regions of the address space.

Our efforts have implications for both the research and operational communities. The ubiquity of background radiation presents significant difficulties for those who monitor Internet traffic: it can clog stateful analyzers with uninteresting activity, and due to its variety it can significantly complicate the detection of new types of activity (for example, a new worm using the same port as existing worms). It is clear from the highly diverse and dynamic activity we have found that further work is needed to both assess the evolution of background radiation over time and to develop more detailed characterizations. We believe that our framework—prefiltering the traffic, using lightweight responders to engage sources in enough detail to categorize them, analyzing the resulting traffic along the axes we have explored—is an important first step towards comprehensively studying this new phenomenon.

Chapter 6

A First Look at Modern Enterprise Traffic

While wide-area Internet traffic has been heavily studied for many years, the characteristics of traffic *inside* Internet enterprises remain almost wholly unexplored. Nearly all of the studies of enterprise traffic available in the literature are well over a decade old and focus on individual LANs rather than whole sites. In this paper we present a broad overview of internal enterprise traffic recorded at a medium-sized site. The packet traces span more than 100 hours, over which activity from a total of several thousand internal hosts appears. This wealth of data—which we have publicly released in anonymized form—spans a wide range of dimensions. While we cannot form general conclusions using data from a single site, and clearly this sort of data merits additional in-depth study in a number of ways, in this chapter we endeavor to characterize a number of the most salient aspects of the traffic. Our goal is to provide a first sense of ways in which modern enterprise traffic is similar to wide-area Internet traffic, and ways in which it is quite different.

The general structure of the chapter is as follows. Section 6.1 presents the background and motivation of this study. Section 6.2 gives an overview of the packet traces we gathered for our study. Next, Section 6.3 presents a broad breakdown of the main components of the traffic, while Section 6.4 looks at the locality of traffic sources and destinations. Section 6.5 examines characteristics of the applications that dominate the traffic. Section 6.6 provides an assessment of the load carried by the monitored networks. Section 6.7

offers final thoughts. We note that given the breadth of the topics covered, we have spread discussions of related work throughout the chapter, rather than concentrating these in their own section.

6.1 Problem Statement

When Cáceres captured the first published measurements of a site's wide-area Internet traffic in July, 1989 [16, 17], the entire Internet consisted of about 130,000 hosts [62]. Today, the largest enterprises can have more than that many hosts just by themselves.

It is striking, therefore, to realize that more than 15 years after studies of wide-area Internet traffic began to flourish, the nature of traffic *inside* Internet enterprises remains almost wholly unexplored. The characterizations of enterprise traffic available in the literature are either vintage LAN-oriented studies [42, 38], or, more recently, focused on specific questions such as inferring the roles played by different enterprise hosts [115] or communities of interest within a site [4]. The only broadly flavored look at traffic within modern enterprises of which we are aware is the study of OSPF routing behavior in [107]. Our aim is to complement that study with a look at the make-up of traffic as seen at the packet level within a contemporary enterprise network.

One likely reason why enterprise traffic has gone unstudied for so long is that it is technically difficult to measure. Unlike a site's Internet traffic, which we can generally record by monitoring a single access link, an enterprise of significant size lacks a single choke-point for its internal traffic. For the traffic we study, we primarily recorded it by monitoring (one at a time) the enterprise's two central routers; but our measurement apparatus could only capture two of the 20+ router ports at any one time, so we could not attain any sort of comprehensive snapshot of the enterprise's activity. Rather, we piece together a partial view of the activity by recording a succession of the enterprise's subnets in turn. This piecemeal tracing methodology affects some of our assessments. For instance, if we happen to trace a portion of the network that includes a large mail server, the fraction of mail traffic will be measured as larger than if we monitored a subnet without a mail server, or if we had an ideally comprehensive view of the enterprise's traffic. Throughout the chapter we endeavor to identify such biases as they are observed. While our methodology is definitely imperfect, to collect traces from a site like ours in a comprehensive fashion would require a large infusion of additional tracing resources.

Our study is limited in another fundamental way, namely that all of our data comes from a single site, and across only a few months in time. It has long been established that the wide-area Internet traffic seen at different sites varies a great deal from one site to another [24, 87] and also over time [87, 88], such that studying a single site *cannot* be representative. Put another way, for wide-area Internet traffic, the very notion of "typical" traffic is not well-defined. We would expect the same to hold for enterprise traffic (though this basic fact actually remains to be demonstrated), and therefore our single-site study can at best provide an *example* of what modern enterprise traffic looks like, rather than a general representation. For instance, while other studies have shown peer-to-peer file sharing applications to be in widespread use [106], we observe nearly none of it in our traces (which is likely a result of organizational policy).

Even given these significant limitations, however, there is much to explore in our packet traces, which span more than 100 hours and in total include activity from 8,000 internal addresses at the Lawrence Berkeley National Laboratory and 47,000 external addresses. Indeed, we found the very wide range of dimensions in which we might examine the data difficult to grapple with. Do we characterize individual applications? Transport protocol dynamics? Evidence for self-similarity? Connection locality? Variations over time? Pathological behavior? Application efficiency? Changes since previous studies? Internal versus external traffic? Etc.

Given the many questions to explore, we decided in this first look project to pursue a broad overview of the characteristics of the traffic, rather than a specific question, with an aim towards informing future, more tightly scoped efforts. To this end, we settled upon the following high-level goals:

1. To understand the makeup (working up the protocol stack from the network layer to the application layer) of traffic on a modern enterprise network.

2. To gain a sense of the patterns of locality of enterprise traffic.

3. To characterize application traffic in terms of how intranet traffic characteristics can differ from Internet traffic characteristics.

4. To characterize applications that might be heavily used in an enterprise network but only rarely used outside the enterprise, and thus have been largely ignored by modeling studies to date.

5. To gain an understanding of the load being imposed on modern enterprise networks.

	D_0	D_1	D_2	D_3	D_4
Date	10/4/04	12/15/04	12/16/04	1/6/05	1/7/05
Duration	10 min	1 hr	1 hr	1 hr	1 hr
Per Tap	1	2	1	1	1-2
# Subnets	22	22	22	18	18
# Packets	17.8M	64.7M	28.1M	21.6M	27.7M
Snaplen	1500	68	68	1500	1500
Mon. Hosts	2,531	2,102	2,088	1,561	1,558
LBNL Hosts	4,767	5,761	5,210	5,234	5,698
Remote Hosts	4,342	10,478	7,138	16,404	23,267

Table 6.1: Dataset characteristics.

Our general strategy in pursuing these goals is "understand the big things first." That is, for each of the dimensions listed above, we pick the most salient contributors to that dimension and delve into them enough to understand their next degree of structure, and then repeat the process, perhaps delving further if the given contributor remains dominant even when broken down into components, or perhaps turning to a different high-level contributor at this point.

6.2 Datasets

We obtained multiple packet traces from two internal network locations at the Lawrence Berkeley National Laboratory (LBNL) in the USA. The tracing machine, a 2.2 GHz PC running FreeBSD 4.10, had four NICs. Each captured a unidirectional traffic stream extracted, via network-controllable Shomiti taps, from one of the LBNL network's central routers. While the kernel did not report any packet-capture drops, our analysis found occasional instances where a TCP receiver acknowledged data not present in the trace, suggesting the reports are incomplete. It is difficult to quantify the significance of these anomalies.

We merged these streams based on timestamps synchronized across the NICs using a custom modification to the NIC driver. Therefore, with the four available NICs we could capture traffic for two LBNL subnets. A further limitation is that our vantage point enabled the monitoring of traffic to and from the subnet, but not traffic that remained within the subnet. We used an *expect* script to periodically change the monitored subnets, working through the 18–22 different subnets attached to each of the two routers.

Table 6.1 provides an overview of the collected packet traces. The "per tap" field indicates the number of traces taken on each monitored router port, and *Snaplen* gives the maximum number of bytes captured for each packet. For example, D_0 consists of full-packet traces from each of the 22 subnets monitored once

	D_0	D_1	D_2	D_3	D_4
IP	99%	97%	96%	98%	96%
non-IP	1%	3%	4%	2%	4%
ARP	10%	6%	5%	27%	16%
IPX	80%	77%	65%	57%	32%
Other	10%	17%	29%	16%	52%

Table 6.2: Fraction of packets observed using the given network layer protocol.

for ten minutes at a time, while D_1 consists of one-hour header-only (68 bytes) traces from the 22 subnets, each monitored twice (i.e., two one-hour traces per subnet).

6.3 Broad Traffic Breakdown

We first take a broad look at the protocols present in our traces, examining the network, transport and application layers.

Table 6.2 shows the distribution of "network layer" protocols, i.e., those above the Ethernet link layer. IP dominates, constituting more than 95% of the packets in each dataset, with the two largest non-IP protocols being IPX and ARP; the distribution of non-IP traffic varies considerably across the datasets, reflecting their different subnet (and perhaps time-of-day) makeup.[1]

Before proceeding further, we need to deal with a somewhat complicated issue. The enterprise traces include *scanning traffic* from a number of sources. The most significant of these sources are legitimate, reflecting proactive vulnerability scanning conducted by the site. Including traffic from scanners in our analysis would skew the proportion of connections due to different protocols. And, in fact, scanners can engage services that otherwise remain idle, skewing not only the magnitude of the traffic ascribed to some protocol but also the number of protocols encountered.

In addition to the known internal scanners, we identify additional scanning traffic using the following heuristic. We first identify sources contacting more than 50 distinct hosts. We then determine whether at least 45 of the distinct addresses probed were in ascending or descending order. The scanners we find with this heuristic are primarily external sources using ICMP probes, because most other external scans get blocked by scan filtering at the LBNL border. Prior to our subsequent analysis, we remove traffic from

[1]Hour-long traces we made of ≈ 100 individual hosts (not otherwise analyzed here) have a makeup of 35–67% *non*-IPv4 packets, dominated by *broadcast* IPX and ARP. This traffic is mainly confined to the host's subnet and hence not seen in our inter-subnet traces. However, the traces are too low in volume for meaningful generalization.

	Connections				
	D0/all	D1/all	D2/all	D3/all	D4/all
Total	198133	1391718	563629	881200	1232789
Scan1	11%	0.9%	2%	2%	2%
Scan2	1%	15%	1%	9%	1%
WAN Scanner	5%	0.6%	1%	3%	3%
Non-Scan	82%	84%	96%	86%	93%

Table 6.3: Fractions of scan connections (removed from further analysis)

	D_0	D_1	D_2	D_3	D_4
Bytes (GB)	13.12	31.88	13.20	8.98	11.75
TCP	66%	95%	90%	77%	82%
UDP	34%	5%	10%	23%	18%
ICMP	0%	0%	0%	0%	0%
Conns (M)	0.16	1.17	0.54	0.75	1.15
TCP	26%	19%	23%	10%	8%
UDP	68%	74%	70%	85%	87%
ICMP	6%	6%	8%	5%	5%

Table 6.4: Fraction of connections and bytes utilizing various transport protocols.

sources identified as scanners along with the two internal scanners. Table 6.3 shows that the fraction of connections removed from the traces ranges from 4–18% across the datasets. A more in-depth study of characteristics that the scanning traffic exposes is a fruitful area for future work.

We now turn to Table 6.4, which breaks down the traffic by transport protocol (i.e., above the IP layer) in terms of payload bytes and packets for the three most popular transports found in our traces. The "Bytes" and "Conns" rows give the total number of payload bytes and connections for each dataset in Gbytes and millions, respectively. The ICMP traffic remains fairly consistent across all datasets, in terms of fraction of both bytes and connections. The mix of TCP and UDP traffic varies a bit more. We note that the bulk of the bytes are sent using TCP, and the bulk of the connections use UDP, for reasons explored below. Finally, we observe a number of additional transport protocols in our datasets, each of which make up only a slim portion of the traffic, including IGMP, IPSEC/ESP, PIM, GRE, and IP protocol 224 (unidentified).

Next we break down the traffic by application category. We group TCP and UDP application protocols as shown in Table 6.5. The table groups the applications together based on their high-level purpose. We show only those distinguished by the amount of traffic they transmit, in terms of packets, bytes or connections (we omit *many* minor additional categories and protocols). In Section 6.5 we examine the characteristics of a number of these application protocols.

Category	Protocols
backup	Dantz, Veritas, “connected-backup”
bulk	FTP, HPSS
email	SMTP, IMAP4, IMAP/S, POP3, POP/S, LDAP
interactive	SSH, telnet, rlogin, X11
name	DNS, Netbios-NS, SrvLoc
net-file	NFS, NCP
net-mgnt	DHCP, ident, NTP, SNMP, NAV-ping, SAP, NetInfo-local
streaming	RTSP, IPVideo, RealStream
web	HTTP, HTTPS
windows	CIFS/SMB, DCE/RPC, Netbios-SSN, Netbios-DGM
misc	Steltor, MetaSys, LPD, IPP, Oracle-SQL, MS-SQL

Table 6.5: Application categories and their constituent protocols.

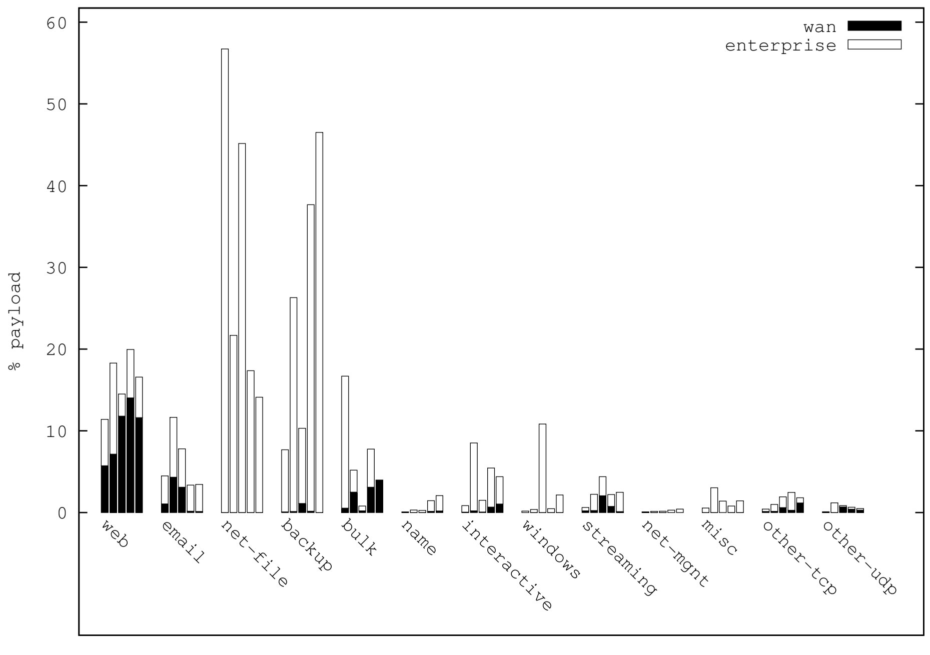


Figure 6.1: Fraction of traffic payload bytes of various application layer protocols.

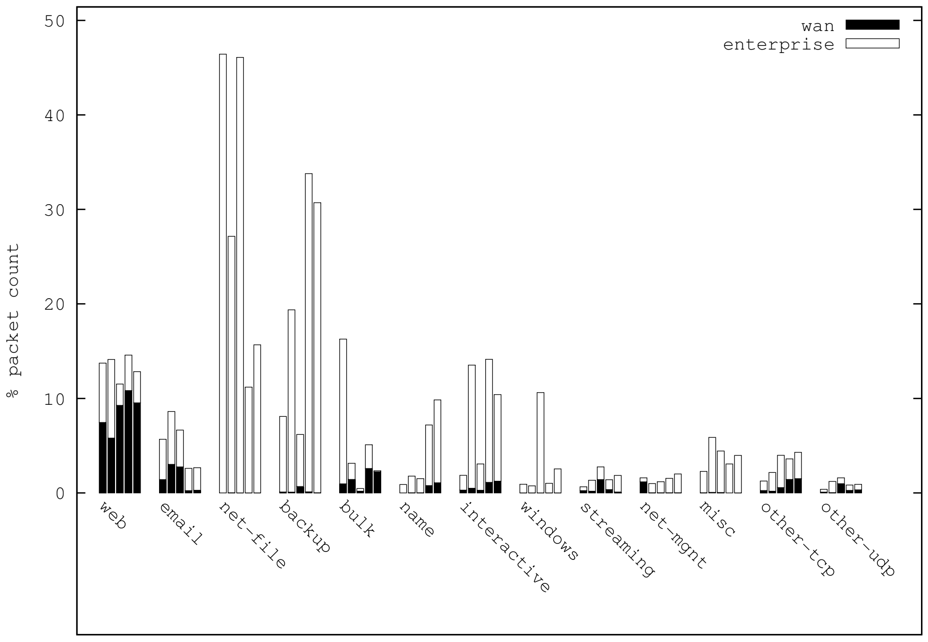


Figure 6.2: Fraction of packets of various application layer protocols.

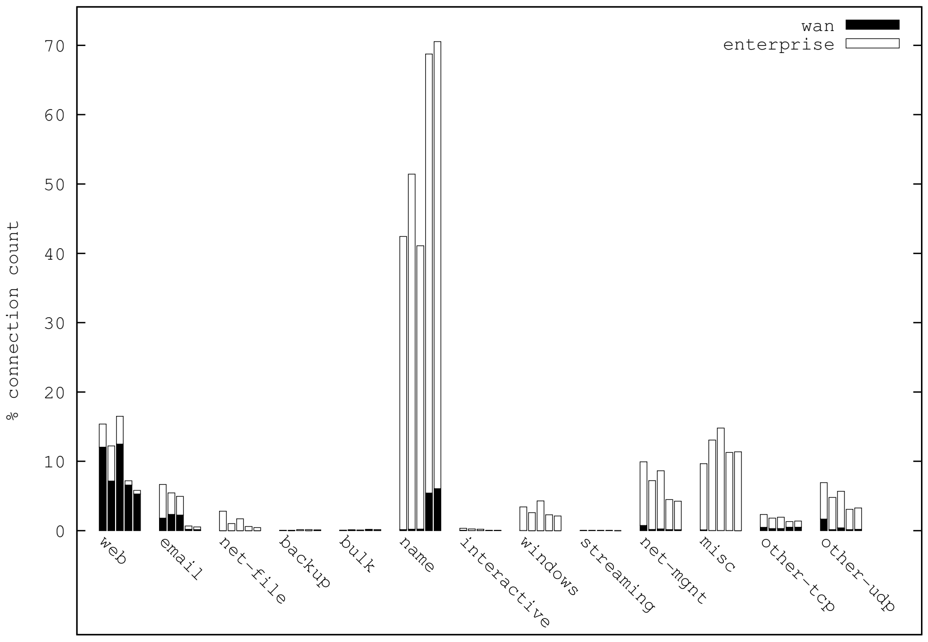


Figure 6.3: Fraction of connections of various application layer protocols.

Figure 6.1, 6.2, and 6.3 show the fraction of unicast payload bytes, packets, and connections from each application category, respectively (multicast traffic is discussed below). The five bars for each category correspond to our five datasets. The total height of the bar represents the percentage of traffic due to the given category. For each dataset, the height of bars adds up to 100%. The solid part of the bar represents the fraction of the total in which one of the endpoints of the connection resides outside of LBNL, while the hollow portion of the bar represents the fraction of the total that remains within LBNL's network. (We delve into traffic origin and locality in more depth in Section 6.4.)

First, the plots show a *wider range of application usage* within the enterprise than over the WAN. In particular, we observed 3–4 times as many application categories on the internal network as we did traversing the border to the WAN. The wider range likely reflects the impact of administrative boundaries such as trust domains and firewall rules, and if so should prove to hold for enterprises in general. The figure also shows that the majority of traffic observed is local to the enterprise. This follows the familiar pattern of locality in computer and network systems which, for example, plays a part in memory, disk block, and web page caching.

In addition, Figure 6.3 and 6.1 show the reason for the finding above that most of the connections in the traces use UDP, while most of the bytes are sent across TCP connections. Many connections are for "name" traffic across all the datasets (45–65% of the connections). However, the byte count for "name" traffic constitutes no more than 1% of the aggregate traffic. The "net-mgnt", "misc" and "other-udp" categories show similar patterns. While most of the connections are short transaction-style transfers, most of the bytes that traverse the network are from relatively few connections.

Figure 6.1 shows that the "bulk", "network-file" and "backup" categories constitute a majority of the bytes observed across datasets. In some of the datasets, "windows", "streaming" and "interactive" traffic each contribute 5–10% of the bytes observed, as well. The first two make sense because they include bulk-transfer as a component of their traffic; and in fact interactive traffic does too, in the form of SSH, which can be used not only as an interactive login facility but also for copying files and tunneling other applications.

The breakdown in term of packets (Figure 6.2) is similar to the breakdown in terms of bytes (Figure 6.1), except that when measuring in terms of packets the percentage of interactive traffic is roughly a factor of two more than when assessing the traffic in terms of bytes, indicating that interactive traffic consists, not surprisingly, of small packets [24].

Most of the application categories shown in the breakdown figures are *unbalanced* in that the traffic is dominated by either the connection count or the byte count. The "web" and "email" traffic categories are the exception; they show non-negligible contributions to both the byte and connection counts. We will characterize these applications in detail in Section 6.5, but here we note that this indicates that most of the traffic in these categories consists of connections with modest—not tiny or huge—lengths.

In addition, the plot highlights the differences in traffic profile across time and area of the network monitored. For instance, the number of bytes transmitted for "backup" activities varies by a factor of roughly five from D_0 to D_4. This could be due to differences in the monitored locations, or different tracing times. Given our data collection techniques, we cannot distill trends from the data; however this is clearly a fruitful area for future work. We note that most of the application categories that significantly contribute to the traffic mix show a range of usage across the datasets. However, the percentage of connections in the "net-mgnt" and "misc" categories are fairly consistent across the datasets. This may be because a majority of the connections come from periodic probes and announcements, and thus have a quite stable volume.

Finally, we note that multicast traffic constitutes a significant fraction of traffic in the "streaming", "name", and "net-mgnt" categories. We observe that 5–10% of all TCP/UDP payload bytes transmitted are in multicast streaming—i.e., more than the amount of traffic found in unicast streaming. Likewise, multicast traffic in "name" (SrvLoc) and "net-mgnt" (SAP) each constitutes 5–10% of all TCP/UDP connections. However, multicast traffic in the remaining application categories was found to be negligible.

6.4 Origins and Locality

We next analyze the data to assess both the origins of traffic and the breadth of communications among the monitored hosts. First, we examine the origin of the flows in each dataset, finding that the traffic is clearly dominated by unicast flows whose source and destination are both within the enterprise (71–79% of flows across the five datasets). Another 2–3% of unicast flows originate within the enterprise but communicate with peers across the wide-area network, and 6–11% originate from hosts outside of the enterprise contacting peers within the enterprise. Finally, 5–10% of the flows use multicast sourced from within the enterprise and 4–7% use multicast sourced externally.

We next assess the number of hosts with which each monitored host communicates. For each monitored host H we compute two metrics: (i) *fan-in* is the number of hosts that originate conversations with H, while

(ii) *fan-out* is the number of hosts to which H initiates conversations. We calculate these metrics in terms of both local traffic and wide-area traffic.

Figure 6.4 shows the distribution of fan-in and fan-out for D_2 and D_3. We observe that for both fan-in and fan-out, the hosts in our datasets generally have more peers within the enterprise than across the WAN, though with considerable variability. In particular, one-third to one-half of the hosts have only internal fan-in, and more than half with only internal fan-out—much more than the fraction of hosts with only external peers. This difference matches our intuition that local hosts will contact local servers (e.g., SMTP, IMAP, DNS, distributed file systems) more frequently than requesting services across the wide-area network, and is also consistent with our observation that a wider variety of applications are used only within the enterprise.

While most hosts have a modest fan-in and fan-out—over 90% of the hosts communicate with at most a couple dozen other hosts—some hosts communicate with scores to hundreds of hosts, primarily busy servers that communicate with large numbers of on- and off-site hosts (e.g., mail servers). Finally, the tail of the internal fan-out, starting around 100 peers/source, is largely due to the peer-to-peer communication pattern of SrvLoc (Service Locator Protocol [117]) traffic.

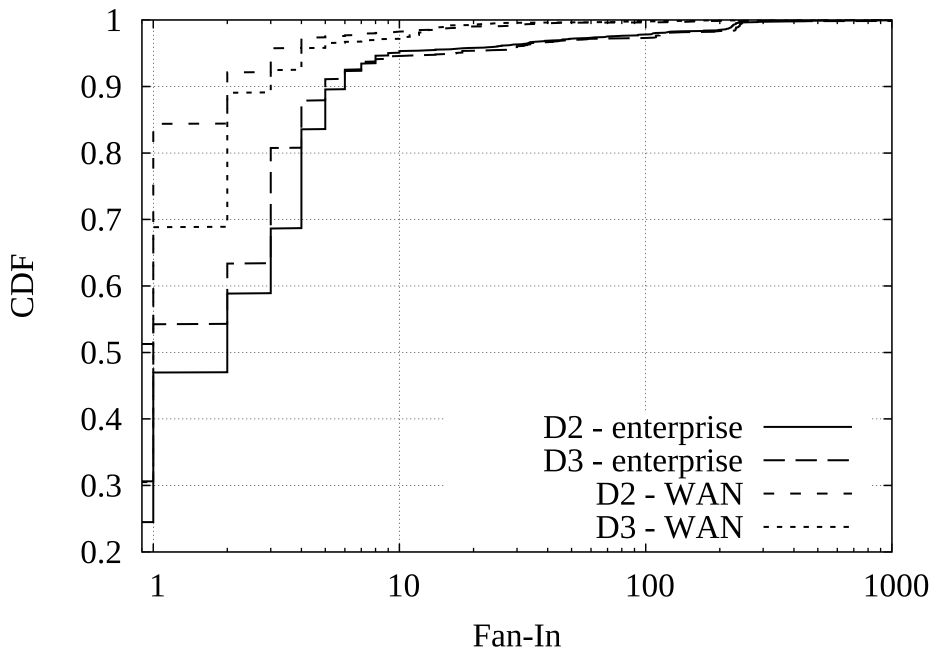
In keeping with the spirit of this study, the data presented in this section provides a first look at origins and locality in the aggregate. Future work on assessing particular applications and examining locality within the enterprise is needed.

6.5 Application Characteristics

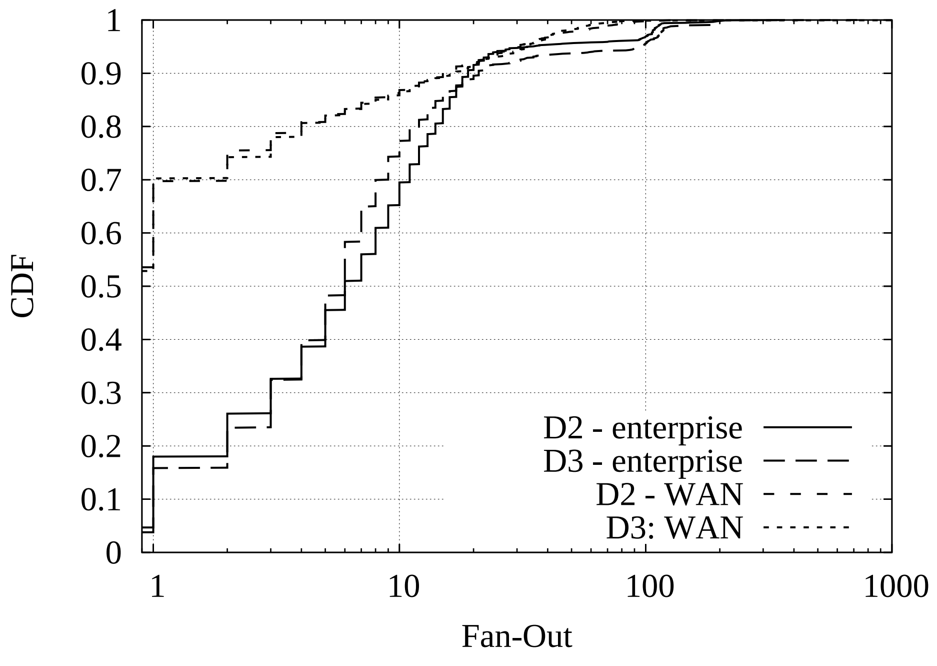
In this section we examine transport-layer and application-layer characteristics of individual application protocols. Table 6.6 provides a number of examples of the findings we make in this section.

We base our analysis on connection summaries generated by Bro [89]. As noted in Section 6.2, D_1 and D_2 consist of traces that contain only the first 68 bytes of each packet. Therefore, we omit these two datasets from analysis that require in-depth examination of packet payloads to extract application-layer protocol messages.

Before turning to specific application protocols, however, we need to first discuss how we compute failure rates. At first blush, counting the number of failed connections/requests seems to tell the story. However, this method can be misleading if the client is automated and endlessly retries after being rejected



(a) Fan-in



(b) Fan-out

Figure 6.4: Locality in host communication.

§ 6.5.1	Automated HTTP client activities constitute a significant fraction of internal HTTP traffic.
§ 6.5.1	IMAP traffic inside the enterprise has characteristics similar to wide-area email, except connections are longer-lived.
§ 6.5.1	Netbios/NS queries fail nearly 50% of the time, apparently due to popular names becoming stale.
§ 6.5.2	Windows traffic is intermingled over various ports, with Netbios/SSN (139/tcp) and SMB (445/tcp) used interchangeably for carrying CIFS traffic. DCE/RPC over "named pipes", rather than Windows File Sharing, emerges as the most active component in CIFS traffic. Among DCE/RPC services, printing and user authentication are the two most heavily used.
§ 6.5.2	Most NFS and NCP requests are reading, writing, or obtaining file attributes.
§ 6.5.2	Veritas and Dantz dominate our enterprise's backup applications. Veritas exhibits only client → server data transfers, but Dantz connections can be large in either direction.

Table 6.6: Example application traffic characteristics.

by a peer, as happens in the case of NCP, for example. Therefore, we instead determine the number of *distinct operations* between *distinct host-pairs* when quantifying success and failure. Such operations can span both the transport layer (e.g., a TCP connection request) and the application layer (e.g., a specific name lookup in the case of DNS). Given the short duration of our traces, we generally find a specific operation between a given pair of hosts either nearly always succeeds, or nearly always fails.

6.5.1 Internal/External Applications

We first investigate applications categories with traffic in both the enterprise network and in the wide-area network: web, email and name service.

Web

HTTP is one of the few protocols where we find more wide-area traffic than internal traffic in our datasets. Characterizing wide-area Web traffic has received much attention in the literature over the years, e.g., [63, 12]. In our first look at modern enterprise traffic, we find internal HTTP traffic to be distinct from WAN HTTP traffic in several ways: (i) we observe that automated clients—scanners, bots, and applications running on top of HTTP—have a large impact on overall HTTP traffic characteristics; (ii) we find a lower fan-out per client in enterprise web traffic than in WAN web traffic; (iii) we find a higher connection failure rate within the enterprise; and (iv) we find heavier use of HTTP's *conditional* `GET` in the internal network than in the WAN. Below we examine these findings along with several additional traffic characteristics.

	Request			Data		
	D0/ent	D3/ent	D4/ent	D0/ent	D3/ent	D4/ent
Total	7098	16423	15712	602MB	393MB	442MB
scan1	20%	45%	19%	0.1%	0.9%	1%
google1	23%	0.0%	1%	45%	0.0%	0.1%
google2	14%	8%	4%	51%	69%	48%
ifolder	1%	0.2%	10%	0.0%	0.0%	9%
All	58%	54%	34%	96%	70%	59%

Table 6.7: Fraction of internal HTTP traffic from automated clients.

Automated Clients: In internal Web transactions we find three activities not originating from traditional user-browsing: *scanners*, *Google bots*, and programs running on top of HTTP (e.g., Novell *iFolder* and Viacom *Net-Meeting*). As Table 6.7 shows, these activities are highly significant, accounting for 34–58% of internal HTTP requests and 59–96% of the internal data bytes carried over HTTP. Including these activities skews various HTTP characteristics. For instance, both Google bots and the scanner have a very high "fan-out"; the scanner provokes many more "404 File Not Found" HTTP replies than standard web browsing; *iFolder* clients use `POST` more frequently than regular clients; and *iFolder* replies often have a uniform size of 32,780 bytes. Therefore, while the presence of these activities is the biggest difference between internal and wide-area HTTP traffic, we exclude these from the remainder of the analysis in an attempt to understand additional differences.

Fan-out: Figure 6.5 shows the distribution of fan-out from monitored clients to enterprise and WAN HTTP servers. Overall, monitored clients visit roughly an order of magnitude more external servers than internal servers. This seems to differ from the finding in Section 6.4 that over all traffic clients tend to access more local peers than remote peers. However, we believe that the pattern shown by HTTP transactions is more likely to be the prevalent application-level pattern and that the results in Section 6.4 are dominated by the fact that clients access a wider variety of applications. This serves to highlight the need for future work to drill down on the first, high-level analysis we present in this study.

Connection Success Rate: Internal HTTP traffic shows success rates of 72–92% (by number of host-pairs), while the success rate of WAN HTTP traffic is 95–99%. The root cause of this difference remains a mystery. We note that the majority of unsuccessful internal connections are terminated with TCP RSTs by the servers, rather than going unanswered.

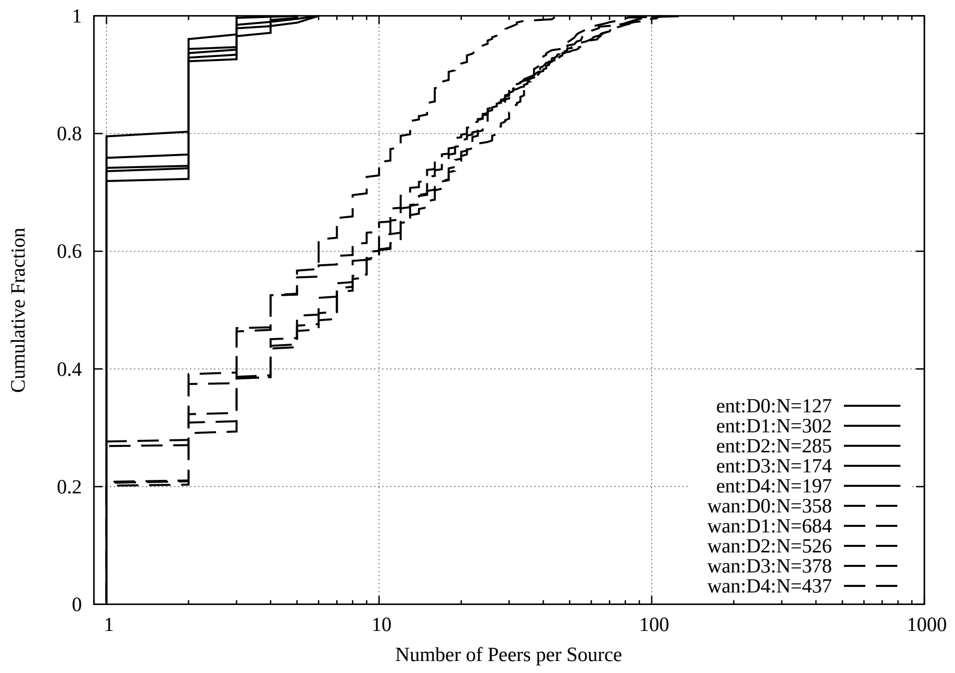


Figure 6.5: HTTP fan-out. The N in the key is the number of samples throughout the chapter—in this case, the number of clients.

Conditional Requests: Across datasets and localities, HTTP `GET` commands account for 95–99% of both the number of requests and the number of data bytes. The `POST` command claims most of the rest. One notable difference between internal and wide area HTTP traffic is the heavier use internally of conditional `GET` commands (i.e., a `GET` request that includes one of the `If-Modified-Since` headers, per [33]). Internally we find conditional `GET` commands representing 29–53% of web requests, while externally conditional `GET` commands account for 12–21% of the requests. The conditional requests often yield savings in terms of the number of data bytes downloaded in that conditional requests only account for 1–9% of the HTTP data bytes transfered internally and 1–7% of the data bytes transfered from external servers. We find this use of the conditional `GET` puzzling in that we would expect that attempting to save wide-area network resources (by caching and only updating content when needed) would be more important than saving local network resources. Finally, we find that over 90% of web requests are successful (meaning either the object requested is returned or that an HTTP 304 (“not modified”) reply is returned in response to a conditional `GET`).

We next turn to several characteristics for which we do not find any *consistent* differences between internal and wide-area HTTP traffic.

	Request		Data	
	enterprise	wan	enterprise	wan
text	18% – 30%	14% – 26%	7% – 28%	13% – 27%
image	67% – 76%	44% – 68%	10% – 34%	16% – 27%
application	3% – 7%	9% – 42%	57% – 73%	33% – 60%
Other	0.0% – 2%	0.3% – 1%	0.0% – 9%	11% – 13%

Table 6.8: HTTP reply by content type. “Other” mainly includes *audio*, *video*, and *multipart*.

Content Type: Table 6.8 provides an overview of object types for HTTP `GET` transactions that received a 200 or 206 HTTP response code (i.e., success). The *text*, *image*, and *application* content types are the three most popular, with *image* and *application* generally accounting for most of the requests and bytes, respectively. Within the *application* type, the popular subtypes include *javascript*, *octet stream*, *zip*, and *PDF*. The *other* content types are mainly *audio*, *video*, or *multipart* objects. We do not observe significant differences between internal and WAN traffic in terms of application types.

HTTP Responses: Figure 6.6 shows the distribution of HTTP response body sizes, excluding replies without a body. We see no significant difference between internal and WAN servers. The short vertical lines of the D_0/WAN curve reflect repeated downloading of javascripts from a particular website. We also find that about half the web sessions (i.e., downloading an entire web page) consist of one object (e.g., just an HTML page). On the other hand 10–20% of the web sessions in our dataset include 10 or more objects. We find no significant difference across datasets or server location (local or remote).

HTTP/SSL: Our data shows no significant difference in HTTPS traffic between internal and WAN servers. However, we note that in both cases there are numerous small connections between given host-pairs. For example, in D_4 we observe 795 short connections between a single pair of hosts during an hour of tracing. Examining a few at random shows that the hosts complete the SSL handshake successfully and exchange a pair of application messages, after which the client tears down the connection almost immediately. As the contents are encrypted, we cannot determine whether this reflects application level fail-and-retry or some other phenomenon.

Email

Email is the second traffic category we find prevalent in both internally and over the wide-area network. As shown in Table 6.9, SMTP and IMAP dominate email traffic, constituting over 94% of the volume in

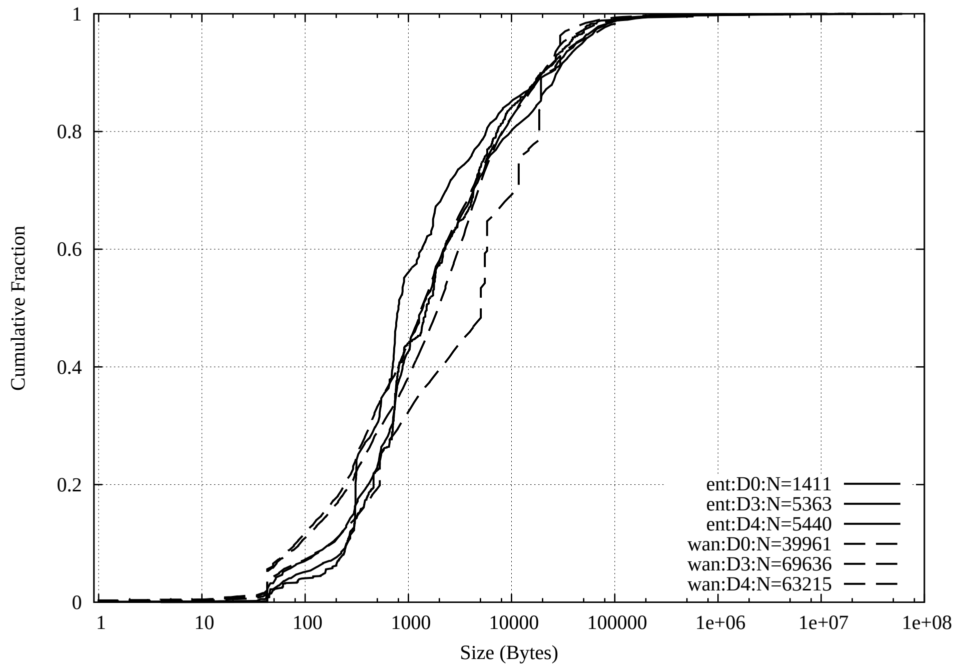


Figure 6.6: Size of HTTP reply, when present.

	Bytes				
	D0/all	D1/all	D2/all	D3/all	D4/all
SMTP	152MB	1658MB	393MB	20MB	59MB
SIMAP	185MB	1855MB	612MB	236MB	258MB
IMAP4	216MB	2MB	0.7MB	0.2MB	0.8MB
Other	9MB	68MB	21MB	12MB	21MB

Table 6.9: Email Traffic Size

bytes. The remainder comes from LDAP, POP3 and POP/SSL. The table shows a transition from IMAP to IMAP/S (IMAP over SSL) between D_0 and D_1, which reflects a policy change at LBNL restricting usage of unsecured IMAP.

Datasets D_{0-2} include the subnets containing the main enterprise-wide SMTP and IMAP(/S) servers. This causes a difference in traffic volume between datasets D_{0-2} and D_{3-4}, and also other differences discussed below. Also, note that we conduct our analysis at the transport layer, since often the application payload is encrypted.

We note that the literature includes several studies of email traffic (e.g., [87, 39]), but none (that we are aware of) focusing on enterprise networks.

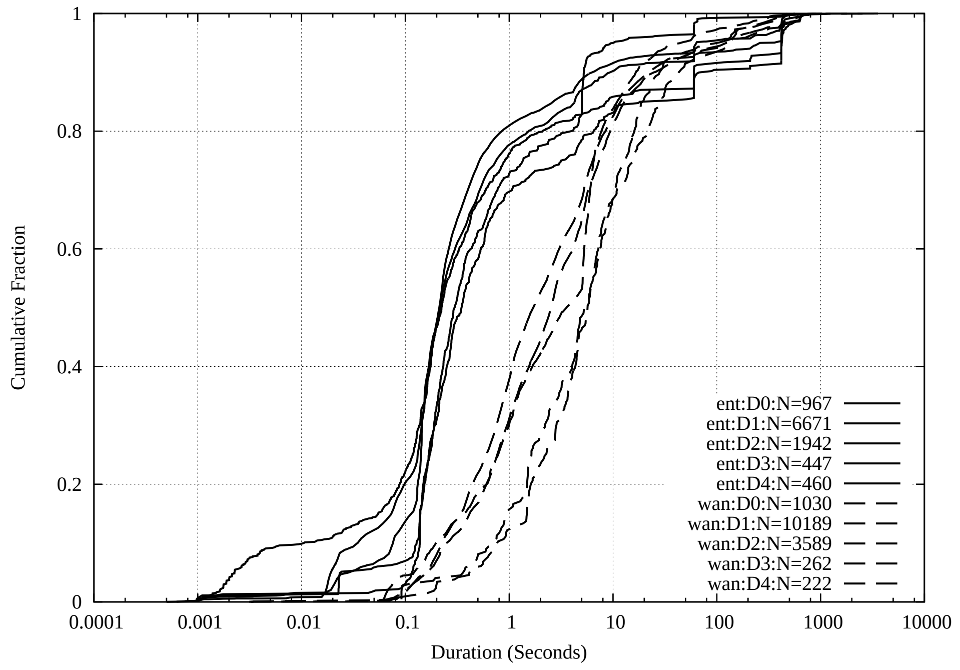
We first discuss areas where we find significant difference between enterprise and wide-area email traffic.

Connection Duration: As shown in Figure 6.7(a), the duration of internal and WAN SMTP connections generally differs by about an order of magnitude, with median durations around 0.2–0.4 sec and 1.5–6 sec, respectively. These results reflect the large difference in round-trip times (RTTs) experienced across the two types of network. SMTP sessions consist of both an exchange of control information and a unidirectional bulk transfer for the messages (and attachments) themselves. Both of these take time proportional to the RTT [76], explaining the longer SMTP durations.

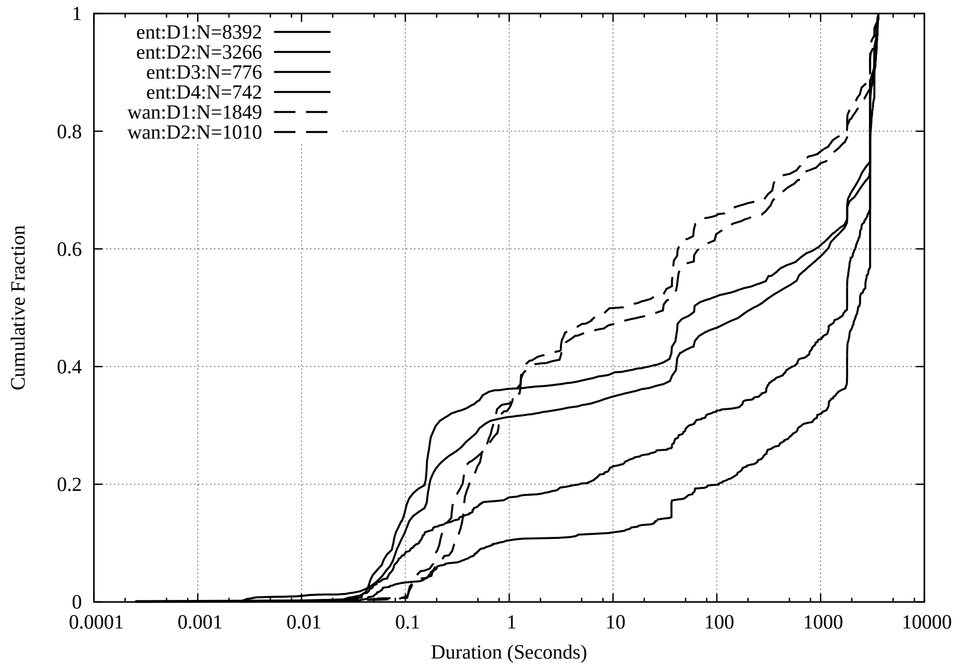
In contrast, Figure 6.7(b) shows the distribution of IMAP/S connection durations across a number of our datasets. We leave off D_0 to focus on IMAP/S traffic, and D_{3-4} WAN traffic because these datasets do not include subnets with busy IMAP/S servers and hence have little wide-area IMAP/S traffic. The plot shows internal connections often last 1–2 orders of magnitude longer than wide-area connections. We do not yet have an explanation for the difference. The maximum connection duration is generally 50 minutes. While our traces are roughly one hour in length we find that IMAP/S clients generally poll every ten minutes, generally providing only five observations within each trace. Determining the true length of IMAP/S sessions requires longer observations and is a subject for future work.

We next focus on characteristics of email traffic that are similar across network type.

Connection Success Rate: Across our datasets we find that internal SMTP connections have success rates of 95–98%. SMTP connections traversing the wide-area network have success rates of 71–93% in D_{0-2} and 99-100% in D_{3-4}. Recall that D_{0-2} include heavily used SMTP servers and D_{3-4} do not, which



(a) SMTP



(b) IMAP/S

Figure 6.7: SMTP and IMAP/S connection durations.

likely explains the discrepancy. The success rate for IMAP/S connections is 99–100% across both locality and datasets.

Flow Size: Internal and wide-area email traffic does not show significant differences in terms of connection sizes, as shown in Figure 6.8. As we would expect, the traffic volume of SMTP and IMAP/S is largely unidirectional (to SMTP servers and to IMAP/S clients), with traffic in the other direction largely being short control messages. Over 95% of the connections to SMTP servers and to IMAP/S clients remain below 1 MB, but both cases have significant upper tails.

Name Services

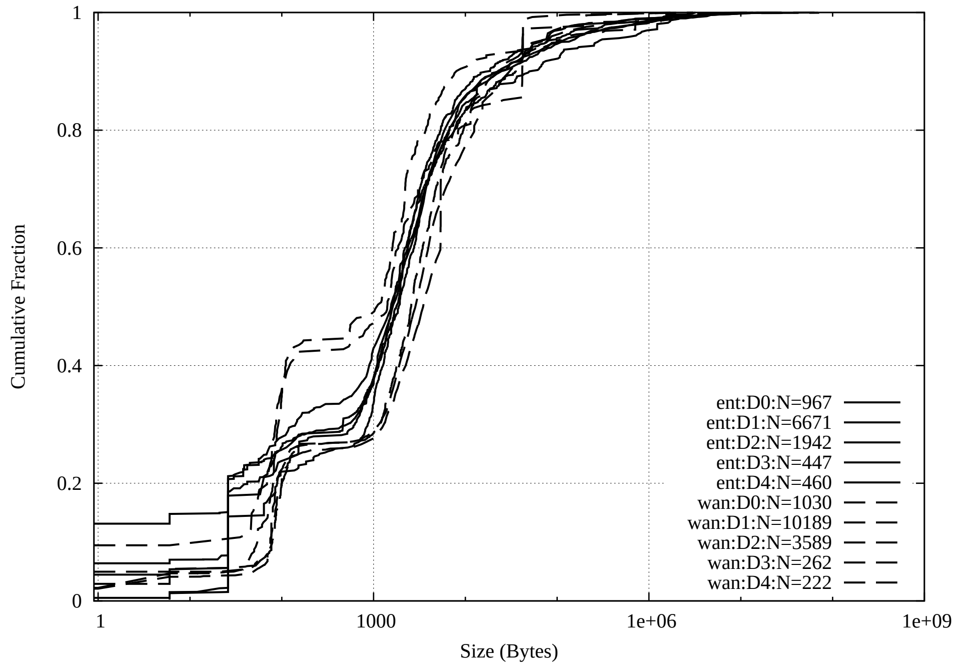
The last application category prevalent in both the internal and the wide-area traffic is domain name lookups. We observe a number of protocols providing name/directory services, including DNS, Netbios Name Service (Netbios/NS), Service Location Protocol (SrvLoc) [117], SUN/RPC Portmapper, and DCE/RPC endpoint mapper. We also note that wide-area DNS has been studied by a number of researchers previously (e.g., [53]), however, our study of name lookups includes both enterprise traffic and non-DNS name services.

In this section we focus on DNS and Netbios/NS traffic, due to their predominant use (Netbios/NS is mainly used for identifying hosts, workgroups, and domains in Windows networking for sharing files, printers, and other services). DNS appears in both wide-area and internal traffic. We find no large differences between the two types of DNS traffic except in response latency.

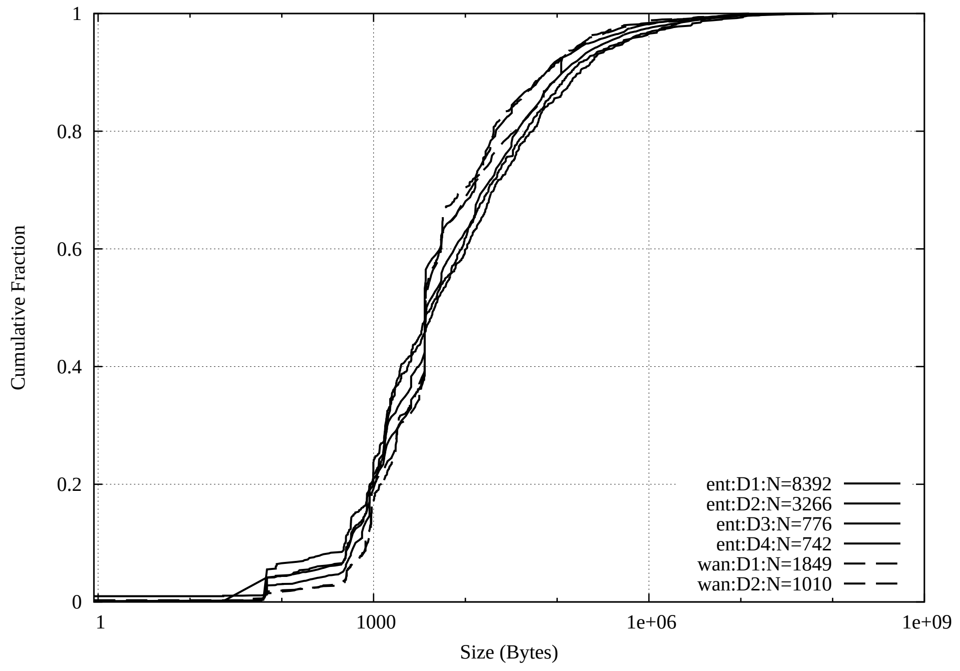
For both services a handful of servers account for most of the traffic, therefore the vantage point of the monitor can significantly affect the traffic we find in a trace. In particular, D_{0-2} do not contain subnets with a main DNS server, and thus relatively few WAN DNS connections. Therefore, in the following discussion we only use D_{3-4} for WAN DNS traffic. Similarly, more than 95% of Netbios/NS requests go to one of the two main servers. D_{0-2} captures all traffic to/from one of these and D_{3-4} captures all traffic to both. Finally, we do not consider D_{1-2} in our analysis due to the lack of application payloads in those datasets (which renders our payload analysis inoperable).

Given those considerations, we now explore several characteristics of name service traffic.

Latency: We observe median latencies are roughly 0.4 msec and 20 msec for internal and external DNS queries, respectively. This expected result is directly attributable to the round-trip delay to on- and off-site



(a) SMTP from client



(b) IMAP/S from server

Figure 6.8: SMTP and IMAP/S: flow size distribution

DNS servers. Netbios/NS, on the other hand, is primarily used within the enterprise, with inbound requests blocked by the enterprise at the border.

Clients: A majority of DNS requests come from a few clients, led by two main SMTP servers that perform lookups in response to incoming and outgoing mail. In contrast, we find Netbios/NS requests more evenly distributed among clients, with the top ten clients generating less than 40% of all requests across datasets.

Request Type: DNS request types are quite similar both across datasets and location of the peer (internal or remote). A majority of the requests (50–66%) are for `A` records, while 17–25% are for `AAAA` (IPv6) records, which seems surprisingly high, though we have confirmed a similar ratio in the wide-area traffic at another site. Digging deeper reveals that a number of hosts are configured to request both `A` and `AAAA` in parallel. In addition, we find 10–18% of the requests are for `PTR` records and 4–7% are for `MX` records.

Netbios/NS traffic is also quite similar across the datasets. 81–85% of requests consist of name queries, with the other prevalent action being to "refresh" a registered name (12–15% of the requests). We observe a number of additional transaction types in small numbers, including commands to register names, release names, and check status.

Netbios/NS Name Type: Netbios/NS includes a "type" indication in queries. We find that across our datasets 63–71% of the queries are for workstations and servers, while 22–32% of the requests are for domain/browser information.

Return Code: We find DNS has similar success (`NOERROR`) rates (77–86%) and failure (`NXDOMAIN`) rates (11–21%) across datasets and across internal and wide-area traffic. We observe failures with Netbios/NS 2–3 times more often: 36–50% of distinct Netbios/NS queries yield an `NXDOMAIN` reply. These failures are broadly distributed—they are not due to any single client, server, or query string. We speculate that the difference between the two protocols may be attributable to DNS representing an administratively controlled name space, while Netbios/NS uses a more distributed and loosely controlled mechanism for registering names, resulting in Netbios/NS names going "out-of-date" due to timeouts or revocations.

6.5.2 Enterprise-Only Applications

The previous subsection deals with application categories found in both internal and wide-area communication. In this section, we turn to analyzing the high-level and salient features of applications used only within the enterprise.

Windows Services

We first consider those services used by Windows hosts for a wide range of tasks, such as Windows file sharing, authentication, printing, and messaging. In particular, we examine Netbios Session Services (SSN), the Common Internet File System (SMB/CIFS), and DCE/RPC. We do not tackle the Netbios Datagram Service since it appears to be largely used *within* subnets (e.g., for "Network Neighborhoods"), and does not appear much in our datasets; and we cover Netbios/NS in Section 6.5.1.

As demonstrated in Figure 6.9, one of the main challenges in analyzing Windows traffic is that each communication scheme can be used in a variety of ways. For instance, TCP port numbers reveal little about the actual application: services can be accessed via multiple channels, and a single port can be shared by a variety of services. Hosts appear to interchangeably use CIFS via its well-known TCP port of 445 and via layering on top of Netbios/SSN (TCP port 139). Similarly, we note that DCE/RPC clients have two ways to find services: (i) using "named pipes" on top of CIFS (which may or may not be layered on top of Netbios/SSN) and (ii) on top of standard TCP and UDP connections without using CIFS, in which case clients consult the Endpoint Mapper to discover the port of a particular DCE/RPC service. Thus, in order to understand the Windows traffic we had to develop rich Bro protocol analyzers, and also merge activities from different transport layer channels. With this in place, we could then analyze various facets of the activities according to application functionality, as follows.

Connection Success Rate: As shown in Table 6.10, we observe a variety of connection success rates for different kinds of traffic: 82–92% for Netbios/SSN connections, 99–100% for Endpoint Mapper traffic, and a strikingly low 46–68% for CIFS traffic. For both Netbios/SSN and CIFS traffic we find the failures are not caused by a few erratic hosts, but rather are spread across hundreds of clients and dozens of servers. Further investigation reveals most of CIFS connection failures are caused by a number of clients connecting to servers on both the Netbios/SSN (139/tcp) and CIFS (445/tcp) port *in parallel*—since the two ports can be used interchangeably. The apparent intention is to use whichever port works while not incurring the cost

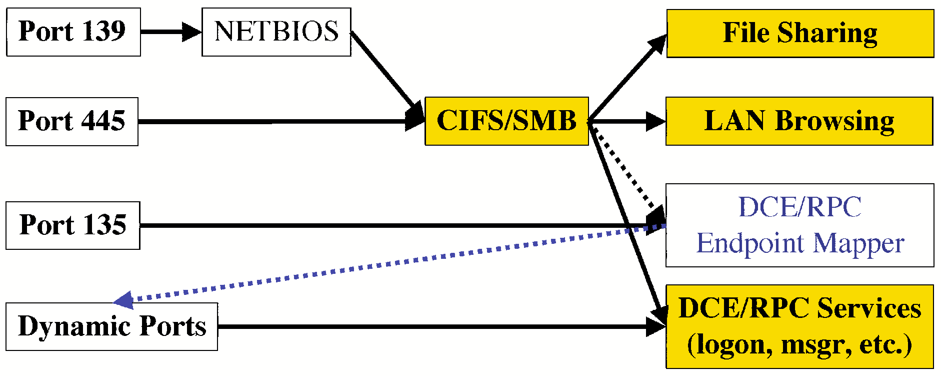


Figure 6.9: Windows (CIFS and DCE/RPC) Traffic Structure

	Host Pairs		
	Netbios/SSN	CIFS	Endpoint Mapper
Total	595 – 1464	373 – 732	119 – 497
Successful	82% – 92%	46% – 68%	99% – 100%
Rejected	0.2% – 0.8%	26% – 37%	0.0% – 0.0%
Unanswered	8% – 19%	5% – 19%	0.2% – 0.8%

Table 6.10: Windows traffic connection success rate (by number of host-pairs, for internal traffic only)

of trying each in turn. We also find a number of the servers are configured to listen only on the Netbios/SSN port, so they reject connections to the CIFS port.

Netbios/SSN Success Rate: After a connection is established, a Netbios/SSN session goes through a handshake before carrying traffic. The success rate of the handshake (counting the number of distinct host-pairs) is 89–99% across our datasets. Again, the failures are not due to any single client or server, but are spread across a number of hosts. The reason for these failures merits future investigation.

CIFS Commands: Table 6.11 shows the prevalence of various types of commands used in CIFS channels across our datasets, in terms of both the number of commands and volume of data transferred. The first category, “SMB Basic”, includes common commands used for session initialization and termination, and accounts for 24–52% of the messages across the datasets, but only 3–15% of the data bytes. The remaining categories indicate the tasks CIFS connections are used to perform. Interestingly, we find DCE/RPC pipes, rather than Windows File Sharing, make up the largest portion of messages (33–48%) and data bytes (32–77%) across datasets. Windows File Sharing constitutes 11–27% of messages and 8% to 43% of bytes.

	Request			Data		
	D0/ent	D3/ent	D4/ent	D0/ent	D3/ent	D4/ent
Total	49120	45954	123607	18MB	32MB	198MB
SMB Basic	36%	52%	24%	15%	12%	3%
RPC Pipes	48%	33%	46%	32%	64%	77%
Windows File Sharing	13%	11%	27%	43%	8%	17%
LANMAN	1%	3%	1%	10%	15%	3%
Other	2%	0.6%	1.0%	0.2%	0.3%	0.8%

Table 6.11: CIFS command breakdown. “SMB basic” includes the common commands shared by all kinds of higher level applications: protocol negotiation, session setup/tear-down, tree connect/disconnect, and file/pipe open.

Finally, “LANMAN”, a non-RPC named pipe for management tasks in “network neighborhood” systems, accounts for just 1–3% of the requests, but 3–15% of the bytes.

DCE/RPC Functions: Since DCE/RPC constitutes an important part of Windows traffic, we further analyze these calls over both CIFS pipes and stand-alone TCP/UDP connections. While we include all DCE/RPC activities traversing CIFS pipes, our analysis for DCE/RPC over stand-alone TCP/UDP connections may be incomplete for two reasons. First, we identify DCE/RPC activities on ephemeral ports by analyzing Endpoint Mapper traffic. Therefore, we will miss traffic if the mapping takes place before our trace collection begins, or if there is an alternate method to discover the server’s ports (though we are not aware of any other such method). Second, our analysis tool currently cannot parse DCE/RPC messages sent over UDP. While this may cause our analysis to miss services that only use UDP, DCE/RPC traffic using UDP accounts for only a small fraction of all DCE/RPC traffic.

Table 6.12 shows the breakdown of DCE/RPC functions. Across all datasets, the *Spoolss* printing functions—and `WritePrinter` in particular—dominate the overall traffic in D_{3-4}, with 63–91% of the requests and 94–99% of the data bytes. In D_0, *Spoolss* traffic remains significant, but not as dominant as user authentication functions (*NetLogon* and *LsaRPC*), which account for 68% of the requests and 52% of the bytes. These figures illustrate the variations present within the enterprise, as well as highlighting the need for multiple vantage points when monitoring. (For instance, in D_0 we monitor a major authentication server, while D_{3-4} includes a major print server.)

	Request			Data		
	D0/ent	D3/ent	D4/ent	D0/ent	D3/ent	D4/ent
Total	14191	13620	56912	4MB	19MB	146MB
NetLogon	42%	5%	0.5%	45%	0.9%	0.1%
LsaRPC	26%	5%	0.6%	7%	0.3%	0.0%
Spoolss/WritePrinter	0.0%	29%	81%	0.0%	80%	96%
Spoolss/other	24%	34%	10%	42%	14%	3%
Other	8%	27%	8%	6%	4%	0.6%

Table 6.12: DCE/RPC function breakdown.

	Connections					Bytes				
	D0/all	D1/all	D2/all	D3/all	D4/all	D0/all	D1/all	D2/all	D3/all	D4/all
NFS	1067	5260	4144	3038	3347	6318MB	4094MB	3586MB	1030MB	1151MB
NCP	2590	4436	2892	628	802	777MB	2574MB	2353MB	352MB	233MB

Table 6.13: NFS/NCP Size

Network File Services

NFS and NCP[2] comprise the two main network file system protocols seen within the enterprise and this traffic is nearly always confined to the enterprise.[3] We note that several trace-based studies of network file system characteristics have appeared in the file system literature (e.g., see [28] and enclosed references). We now investigate several aspects of network file system traffic.

Aggregate Sizes: Table 6.13 shows the number of NFS and NCP connections and the amount of data transferred for each dataset. In terms of connections, we find NFS more prevalent than NCP, except in D_0. In all datasets, we find NFS transfers more data bytes per connection than NCP. As in previous sections, we see the impact of the measurement location in that the relative amount of NCP traffic is much higher in D_{0-2} than in D_{3-4}. Finally, we find "heavy hitters" in NFS/NCP traffic: the three most active NFS host-pairs account for 89–94% of the data transfered, and for the top three NCP host-pairs, 35–62%.

Keep-Alives: We find that NCP appears to use TCP keep-alives to maintain long-lived connections and detect runaway clients. Particularly striking is that 40–80% of the NCP connections across our datasets consist *only* of periodic retransmissions of one data byte and therefore do not include any real activity.

UDP vs. TCP We had expected that NFS-over-TCP would heavily dominate modern use of NFS, but find this is not the case. Across the datasets, UDP comprises 66%/16%/31%/94%/7% of the payload bytes, an

[2]NCP is the *Netware Control Protocol*, a veritable kitchen-sink protocol supporting hundreds of message types, but primarily used within the enterprise for file-sharing and print service.

[3]We found three NCP connections with remote hosts across all our datasets!

	Request			Data		
	D0/ent	D3/ent	D4/ent	D0/ent	D3/ent	D4/ent
Total	697512	303386	607108	5843MB	676MB	1064MB
Read	70%	25%	1%	64%	92%	6%
Write	15%	1%	19%	35%	2%	83%
GetAttr	9%	53%	50%	0.2%	4%	5%
LookUp	4%	16%	23%	0.1%	2%	4%
Access	0.5%	4%	5%	0.0%	0.4%	0.6%
Other	2%	0.9%	2%	0.1%	0.2%	1%

Table 6.14: NFS requests breakdown.

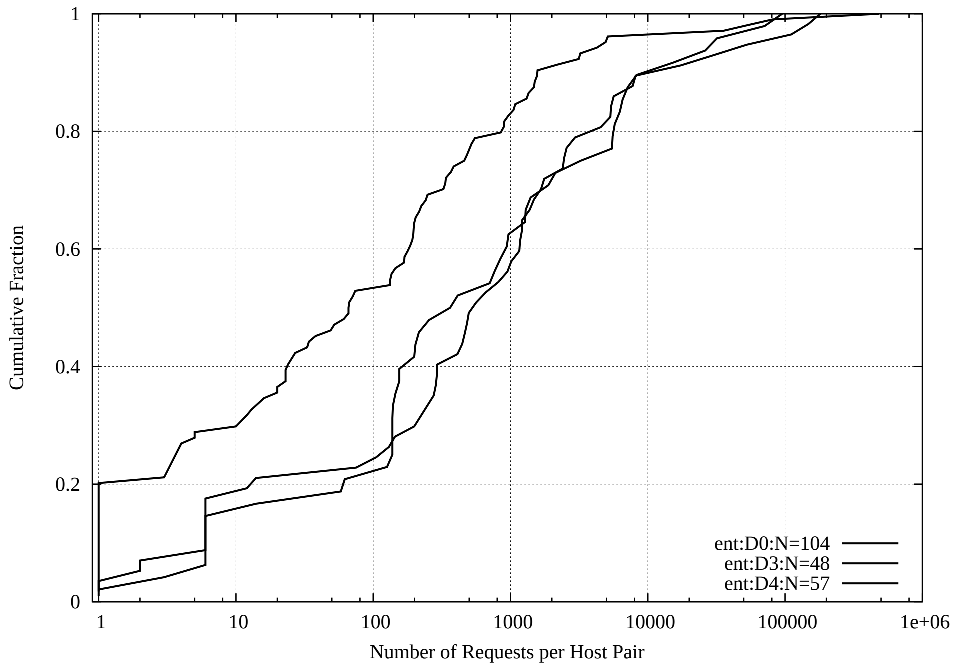
enormous range. Overall, 90% of the NFS host-pairs use UDP, while only 21% use TCP (some use both). One possible explanation is that the enterprise internal network has a low packet drop rate, so there is little incentive to use TCP over UDP.

Request Success Rate: If an NCP connection attempt succeeds (88–98% of the time), about 95% of the subsequent requests also succeed, with the failures dominated by “File/Dir Info” requests. NFS requests succeed 84% to 95% of the time, with most of the unsuccessful requests being “lookup” requests for non-existing files or directories.

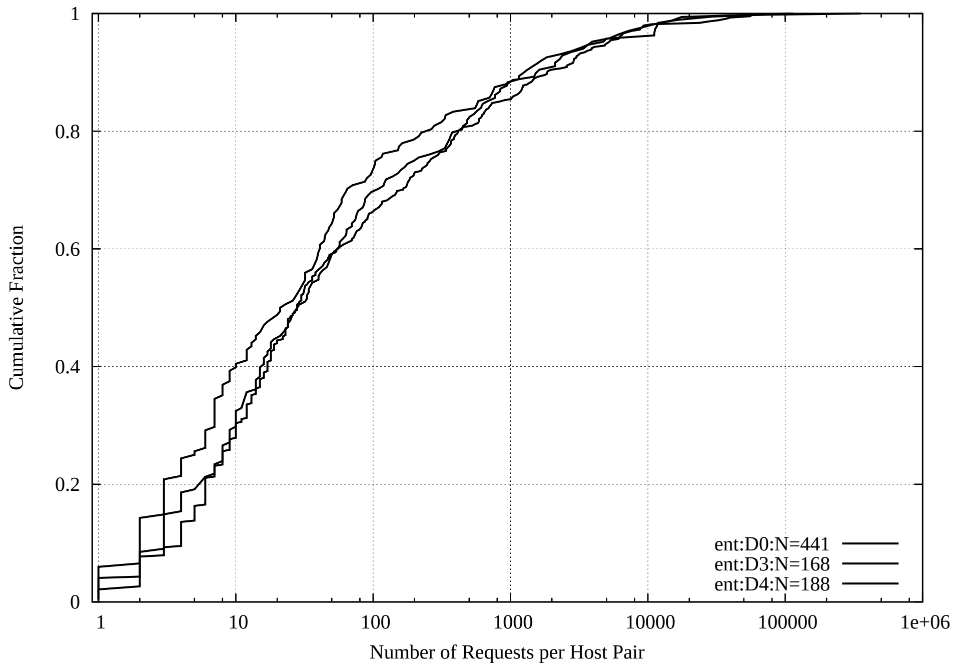
Requests per Host-Pair: Since NFS and NCP both use a message size of about 8 KB, multiple requests are needed for large data transfers. Figure 6.10 shows the number of requests per client-server pair. We see a large range, from a handful of requests to hundreds of thousands of requests between a host-pair. A related observation is that the interval between requests issued by a client is generally 10 msec or less.

Breakdown by Request Type: Table 6.14 and 6.15 show that in both NFS and NCP, file read/write requests account for the vast majority of the data bytes transmitted, 88–99% and 92–98% respectively. In terms of the number of requests, obtaining file attributes joins read and write as a dominant function. NCP file searching also accounts for 7–16% of the requests (but only 1–4% of the bytes). Note that NCP provides services in addition to remote file access, e.g., directory service (NDS), but, as shown in the table, in our datasets NCP is predominantly used for file sharing.

Request/Reply Data Size Distribution: As shown in Figure 6.11, NFS requests and replies have clear dual-mode distributions, with one mode around 100 bytes and the other 8 KB. The latter corresponds to write requests and read replies, and the former to everything else. On the other hand, as Figure 6.12 shows, NCP requests exhibit a mode at 14 bytes, corresponding to read requests, and each vertical rise in the NCP



(a) NFS



(b) NCP

Figure 6.10: NFS/NCP: number of requests per client-server pair, for those with at least one request seen.

	Request			Data		
	D0/ent	D3/ent	D4/ent	D0/ent	D3/ent	D4/ent
Total	869765	219819	267942	712MB	345MB	222MB
Read	42%	44%	41%	82%	70%	82%
Write	1%	21%	2%	10%	28%	11%
FileDirInfo	27%	16%	26%	5%	0.9%	3%
File Open/Close	9%	2%	7%	0.9%	0.1%	0.5%
File Size	9%	7%	5%	0.2%	0.1%	0.1%
File Search	9%	7%	16%	1%	0.6%	4%
Directory Service	2%	0.7%	1%	0.7%	0.1%	0.4%
Other	3%	3%	2%	0.2%	0.1%	0.1%

Table 6.15: NCP requests breakdown.

	Connections	Bytes
VERITAS-BACKUP-CTRL	1271	0.1MB
VERITAS-BACKUP-DATA	352	6781MB
DANTZ	1013	10967MB
CONNECTED-BACKUP	105	214MB

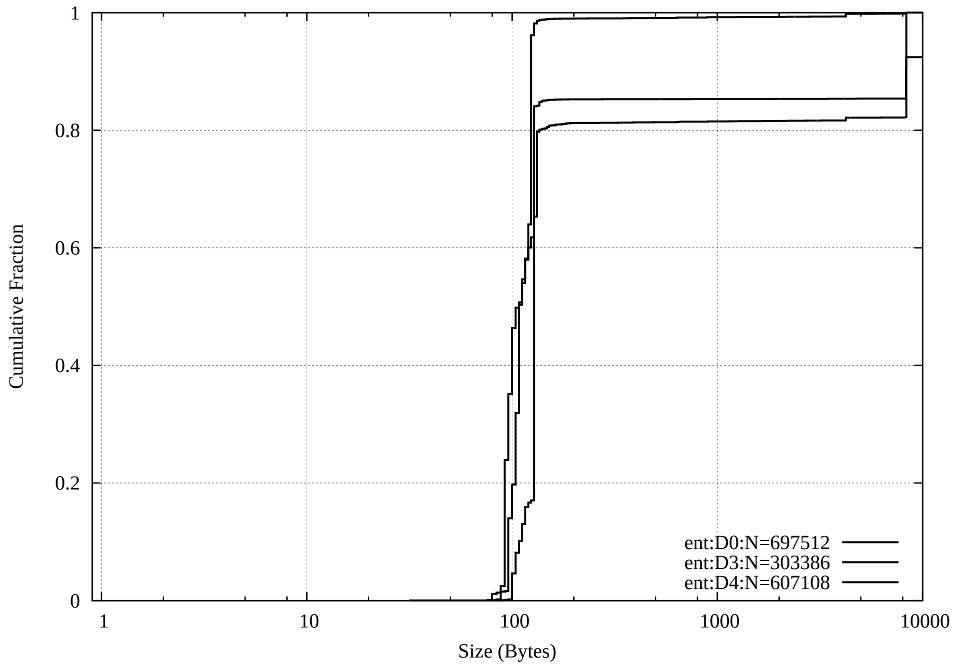
Table 6.16: Backup Applications

reply size figure corresponds to particular types of commands: 2-byte replies for completion codes only (e.g. replying to “WriteFile” or reporting error), 10 bytes for “GetFileCurrentSize”, and 260 bytes for (a fraction of) “ReadFile” requests.

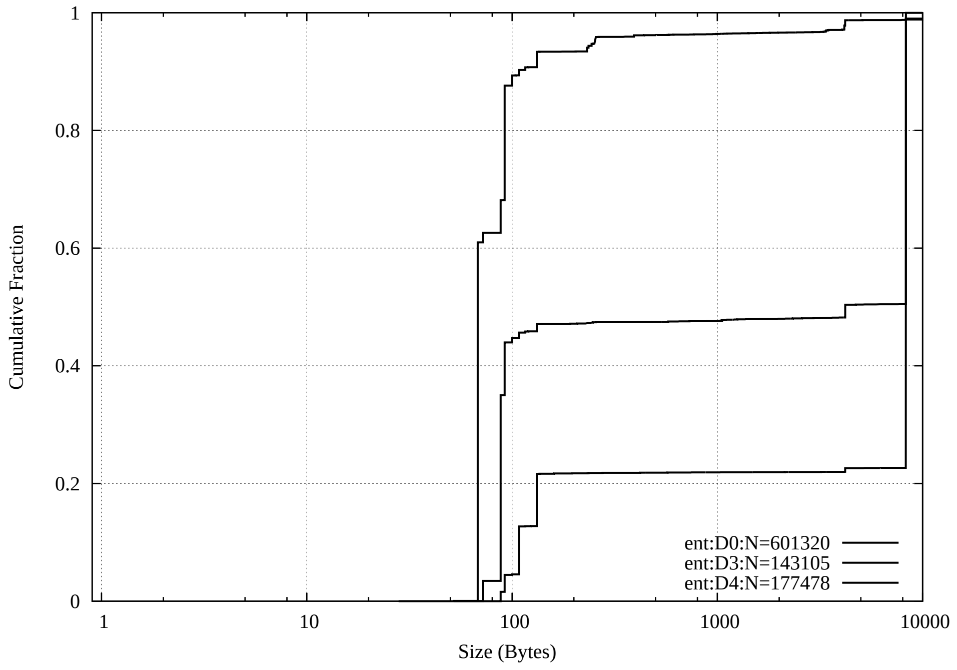
Backup

Backup sessions are a rarity in our traces, with just a small number of hosts and connections responsible for a huge data volume. Clearly, this is an area where we need longer traces. That said, we offer brief characterizations here to convey a sense of its nature.

We find three types of backup traffic, per Table 6.16: two internal traffic giants, Dantz and Veritas, and a much smaller, “Connected” service that backs up data to an external site. Veritas backup uses separate control and data connections, with the data connections in the traces all reflecting one-way, client-to-server traffic. Dantz, on the other hand, appears to transmit control data within the same connection, and its connections display a degree of bi-directionality. Furthermore, the server-to-client flow sizes can exceed 100 MB. This bi-directionality does not appear to reflect backup vs. restore, because it exists not only *between* connections, but also *within* individual connections—sometimes with tens of MB in both directions. Perhaps this reflects an exchange of fingerprints used for compression or incremental backups or an

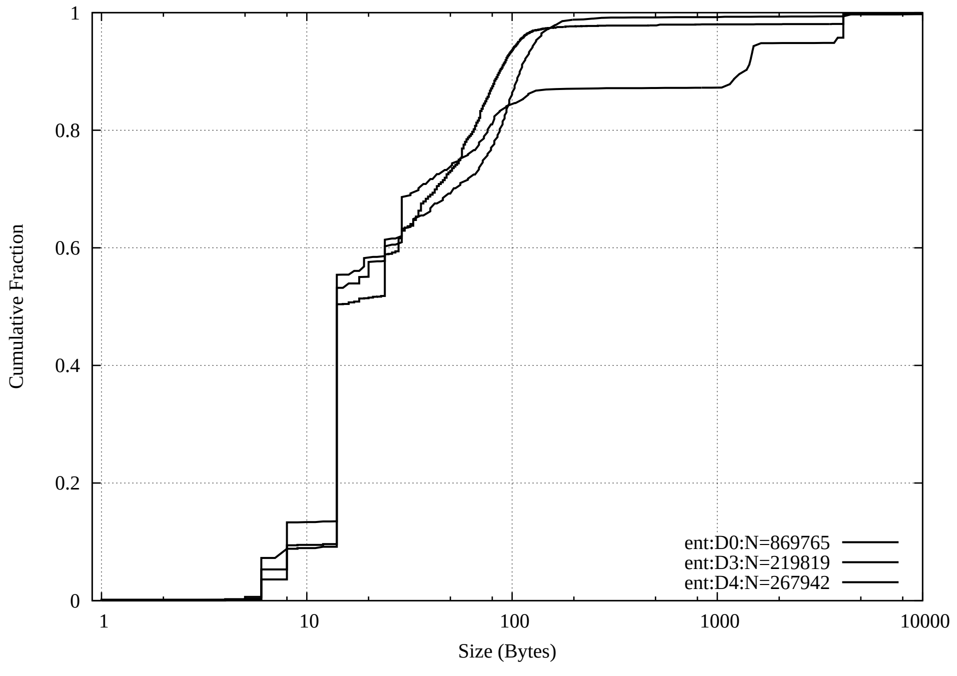


(a) NFS request

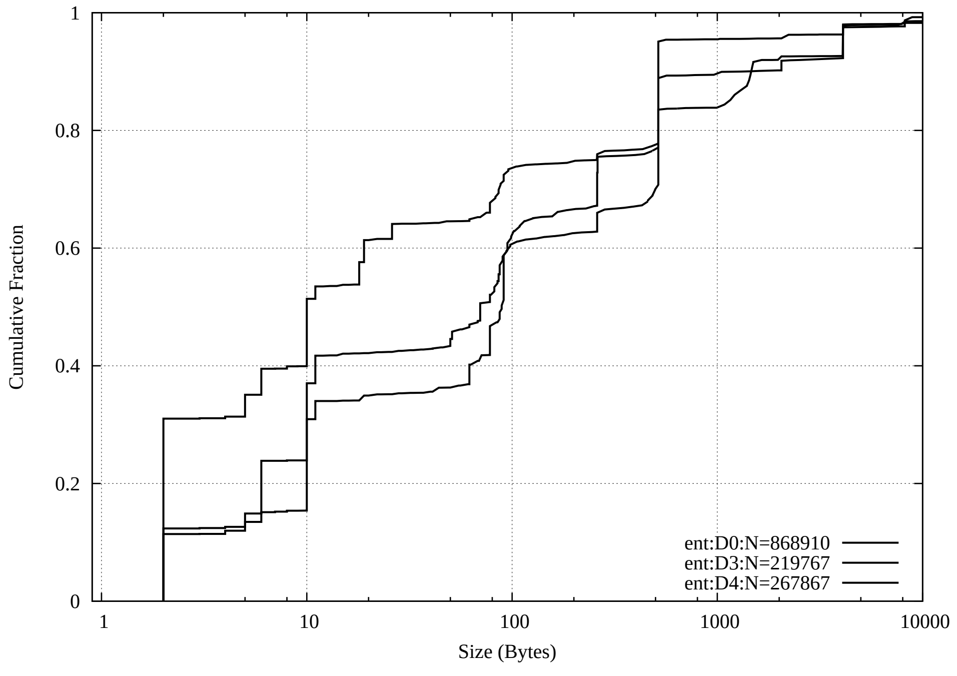


(b) NFS reply

Figure 6.11: NFS: request/reply data size distribution (message headers are not included)



(a) NCP request



(b) NCP reply

Figure 6.12: NCP: request/reply data size distribution (message headers are not included)

exchange of validation information after the backup is finished. Alternatively, this may indicate that the protocol itself may have a peer-to-peer structure rather than a strict server/client delineation. Clearly this requires further investigation with longer trace files.

6.6 Network Load

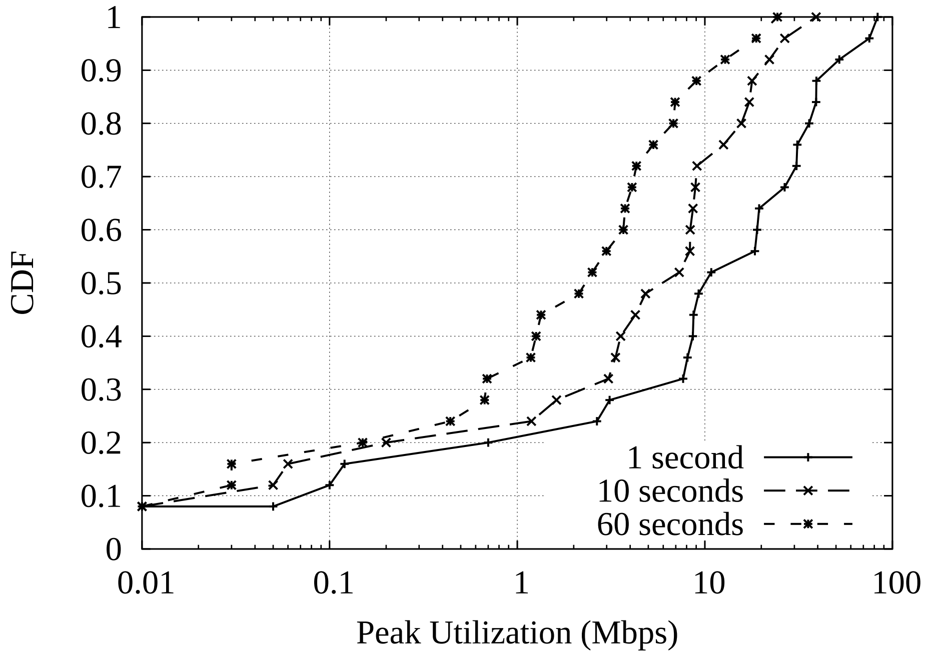
A final aspect of enterprise traffic in our preliminary investigation is to assess the load observed within the enterprise. One might naturally assume that campus networks are underutilized, and some researchers aim to develop mechanisms that leverage this assumption [37]. We assess this assumption using our data.

We discuss only D_4, since the other datasets provide essentially the same insights about utilization. Figure 6.13(a) shows the distribution of the *peak* bandwidth usage over three different timescales for each trace in the D_4 dataset. As expected, the plot shows the networks to be less than fully utilized at each timescale. The one-second interval does show network saturation (100 Mbps) in some cases. However, as the measurement time interval increases the peak utilization drops, indicating that saturation is short-lived.

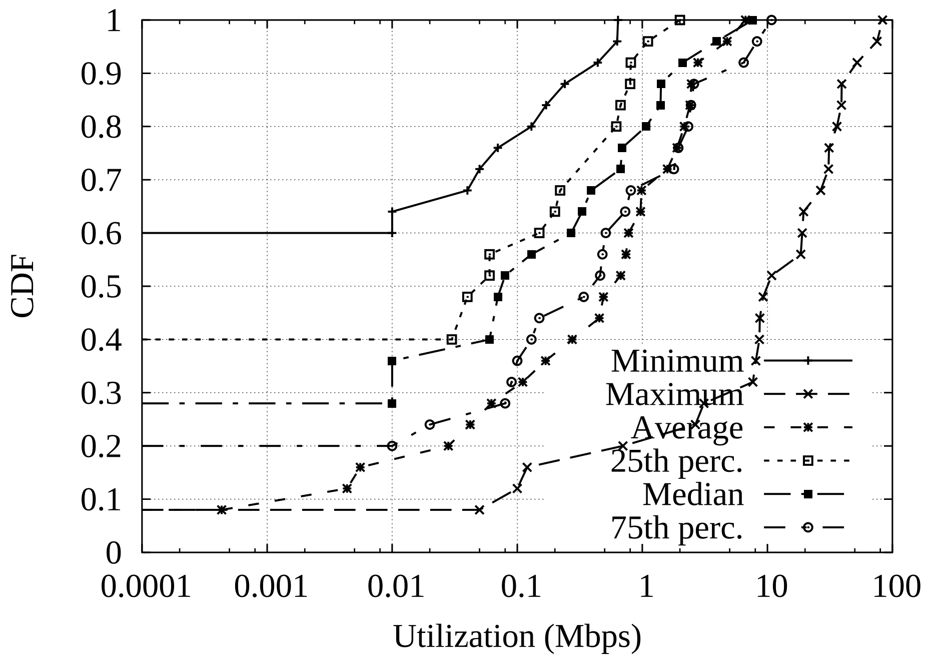
Figure 6.13(b) shows the distributions of several metrics calculated over one-second intervals. The "maximum" line on this plot is the same as the "one-second" line on the previous plot. The second plot concretely shows that typical (over time) network usage is 1–2 orders of magnitude less than the peak utilization and 2–3 orders less than the capacity of the network (100 Mbps).

We can think of packet loss as a second dimension for assessing network load. We can form estimates of packet loss rates using TCP retransmission rates. These two might not fully agree, due to (i) TCP possibly retransmitting unnecessarily, and (ii) TCP adapting its rate in the presence of loss, while non-TCP traffic will not. But the former should be rare in LAN environments (little opportunity for retransmission timers to go off early), and the latter arguably at most limits our analysis to applying to the TCP traffic, which dominates the load (cf. Table 6.4).

We found a number of spurious one-byte retransmissions due to TCP keep-alives by NCP and SSH connections. We exclude these from further analysis because they do not indicate load imposed on network elements. Figure 6.14 shows the remaining retransmission rate for each trace in all our datasets, for both internal and remote traffic. In the vast majority of the traces, the retransmission rate remains less than 1% for both. In addition, the retransmission rate for internal traffic is less than that of traffic involving a remote peer, which matches our expectations since wide-area traffic traverses more shared, diverse, and constrained



(a) Peak Utilization



(b) Utilization

Figure 6.13: Utilization distributions for D_4.

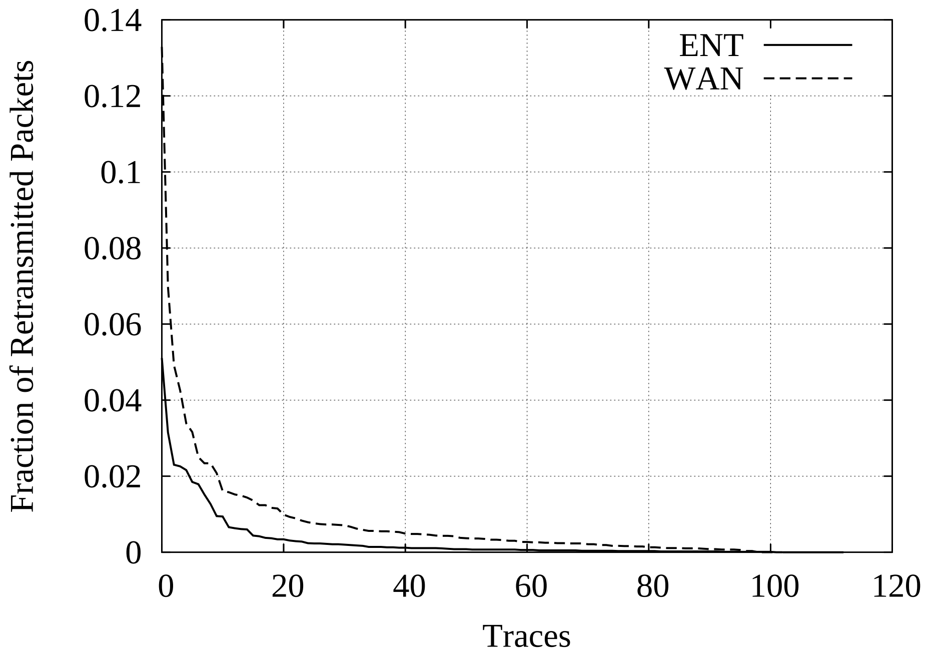


Figure 6.14: TCP retransmission rate across traces (for traces with at least 1000 packets in the category)

networks than does internal traffic. (While not shown in the Figure, we did not find any correlation between internal and wide-area retransmission rates at per-subnet granularity.)

We do, however, find that the internal retransmission rate sometimes eclipses 2%—peaking at roughly 5% in one of the traces. Our further investigation of this last trace found the retransmissions dominated by a single Veritas backup connection, which transmitted 1.5 M packets and 2 GB of data from a client to a server over one hour. The retransmissions happen almost evenly over time, and over one-second intervals the rate peaks at 5 Mbps with a 95^{th} percentile around 1 Mbps. Thus, the losses appear due to either significant congestion in the enterprise network downstream from our measurement point, or a network element with flaky NIC (reported in [113] as not a rare event).

We can summarize these findings as: packet loss within an enterprise appears to occur significantly less than across the wide-area network, as expected; but exceeds 1% a non-negligible proportion of the time.

6.7 Summary

Enterprise networks have been all but ignored in the modern measurement literature. Our major contribution in this study is to provide a broad, high-level view of numerous aspects of enterprise network traffic. Our investigation runs the gamut from re-examining topics previously studied for wide-area traffic (e.g., web traffic), to investigating new types of traffic not assessed in the literature to our knowledge (e.g., Windows protocols), to testing assumptions about enterprise traffic dynamics (e.g., that such networks are mostly underutilized).

Clearly, our investigation is only an initial step in this space. An additional hope for our work is to inspire the community to undertake more in-depth studies of the raft of topics concerning enterprise traffic that we could only examine briefly (or not at all) in this study. Towards this end, we have released anonymized versions of our traces to the community [82].

Chapter 7

Conclusions

This chapter summarizes dissertation contributions and outlines the future directions.

7.1 Contributions

7.1.1 The `binpac` Language

The first contribution of this dissertation is the design and implementation of `binpac`, a declarative language for generating protocol parsers from high-level specifications. Prior to this work network protocol parsers have been mostly hand-written, requiring considerable and often tedious efforts. We envision `binpac` to serve in a role analogous to that of `yacc`'s for generating parsers for programming languages. We note, however, that network protocols are different from programming languages in a number of ways, in particular, network traffic consists of many continuous and bi-directional flows. Consequently the design of `binpac` has a few key features that distinguish it from systems, such as `yacc`, for generating programming language parsers. `binpac` introduces *parameterized types*—a limited form of context-sensitive grammar—to allow concise description of data dependency across network messages and flows. This simple feature allows `binpac` to describe a wide range of protocols. The language also includes the concepts of connection and flow as state container in parsing network traffic. The `binpac` compiler automates saving states for later continuation in parsers to enable switching between flows within a single processor thread, thus supporting efficient processing of a large number of concurrent flows without requiring large number of threads. Finally, the design of `binpac` supports separate specification of the intrinsic syntax and semantics of the protocol from specification of custom processing (for example, Bro event generation), therefore allowing the basic protocol specification part to be *reused* across multiple systems. Our evaluation shows that parser specifications in `binpac` are half to two-third shorter than the corresponding parsers handwritten in C++. On the other hand, the `binpac` compiler generates efficient protocol parsers in C++, whose performance is comparable to handcrafted parsers.

7.1.2 Semantic-Aware Traffic Anonymization

The second contribution of this dissertation lies in our efforts of application level traffic anonymization. Anonymization of network traces is critical for creating public repositories of network traffic. Prior to our attempt, there had been no publicly available packet traces with application level payloads—to the best of our knowledge, largely because of the difficulty of anonymization. Lack of such traces significantly hinders research that relies on semantic traffic analysis, in particular, network intrusion detection.

We note that there are two main problems in traffic anonymization: (1) difficulty in rewriting packet traces due to lack of application-level abstraction for traffic modification, (2) lack of understanding on how to anonymize application level data, which has much richer semantics than TCP/IP fields have. This dissertation addresses both problems. For the first problem, it presents a trace rewriting framework—on top of the `bro` network intrusion detection system [89, 86]—that greatly simplifies application level data manipulation. Within the framework, a rewriting script can focus on application level semantics, for example, how to transform a FTP user name, while the underlying mechanisms make the corresponding changes in both application protocol and TCP/IP (such as TCP sequence number and TCP and IP checksums) to reflect the change. Therefore this framework provides a foundation for application level trace anonymization efforts. For the second problem, this dissertation explores issues in anonymizing FTP traces collected at LBNL and releasing them to the public. It proposes the "filter-in" principle in trace anonymization to safeguard incomplete coverage. It also investigates common forms of inference attacks on trace privacy and how to defend against them. Our treatment of various types of semantic elements, especially identity data, provides a foundation for future efforts in application level traffic anonymization.

7.1.3 Internet Background Radiation

The third contribution of this dissertation is our study of *Internet background radiation*—a special kind of traffic that universally reaches any Internet address, including unused ones. Ours is the first broad characterization of Internet background radiation.

We develop methods to observe and analyze Internet background radiation through large *network telescopes*—blocks of unused, but globally reachable IP addresses. We build application protocol responders to keep conversation with sources going as far as possible, till their intention is revealed. We apply destination-per-source filtering to reduce the traffic volume. Finally, we use `bro` for automated traffic semantic analysis. We find that the majority of Internet background radiation is generated by malicious programs, including Internet worms, that attempt to propagate by randomly searching for vulnerable victims. Our classification of their behavior at application level reveals the rich collection of different kinds of malicious exploits. The study also characterizes the difference of background radiation observed across telescopes at different locations and over time.

7.1.4 Enterprise Network Traffic

The fourth and final contribution of this dissertation is our characterization of modern enterprise network traffic. Unlike wide-area Internet traffic, enterprise internal network traffic has been almost unstudied in the past decade. Based on traces collected at LBNL's internal network, our study presents a first look at the enterprise traffic to provide a sense of ways in which modern enterprise traffic is similar to wide-area Internet traffic, and ways in which it is quite different. The study reveals that the enterprise internal traffic comes from a much richer collection of network applications than wide-area traffic coming into or going out of the enterprise network does; the internal traffic volume is dominated by network file systems and back-up traffic. Furthermore, zooming into individual applications, the study re-examines applications well-studied for wide-area traffic (for example, Web and DNS traffic), and investigates new types of traffic not assessed in literature (for instance, Windows traffic).

7.2 Future Directions

Semantic traffic analysis is a new research area with a lot of questions to be explored. An important part of future exploration in this area will lie in *using* semantic traffic analysis to understand application level characteristics of specific kinds of network traffic, similar to the two traffic studies presented in this dissertation. On the other hand, there is also a lot of room for improvement in the art of semantic traffic analysis and anonymization—which we discuss in further details in this section.

7.2.1 Traffic Analysis

Traffic analyzers in custom hardware. One important future direction in semantic traffic analysis is to explore fundamentally new ways to program and structure network traffic analysis systems so that we can leverage massive parallelism available on future computing hardware. As discussed in further details in our HotSec'06 paper [85], today's network security systems are facing fiercely growing performance pressure from a number of trends. First, as intrusions become increasingly profit-driven and their techniques ever more stealthy and sophisticated, network security systems need to conduct increasingly complex analysis—processing traffic at higher semantic levels and incorporating context correlated across multiple connections, hosts, sensors, and over time. Further, the security systems also need to *rewrite* traffic on the fly (1) to eliminate broad classes of evasion threats ("normalization") and (2) to *prevent* attacks rather than passively "detecting" them. Thus the systems need to analyze traffic in real-time and alter it *inline* without introducing excessive latency. Finally, the growth of network traffic volume has been outpacing that of single-processor performance, and this is exacerbated by the collapse of Moore's law for processor clock rate. While the main alternatives—ASICs and FPGAs—support vastly more processing power through parallelism, they require highly deliberate, customized programming, which is directly at odds with the pressing need to perform diverse and rich forms of analysis. Given these growing pressures—more sophisticated forms of analysis, conducted inline, at higher rates, on non-uniprocessor hardware—it is time to fundamentally rethink the ways of using hardware to support network security analysis. The key, we think, is to devise an abstraction of parallel processing that allows us to expose parallelism latent in semantically rich, stateful analysis algorithms and that we can further compile to hardware platforms with different capabilities.

Parallelization is a known hard problem. But in the particular case of network security analysis, there is vast amount of potential parallelism that can be exploited if we structure and express the tasks properly. Figure 7.1 shows the spectrum of parallelism present in a high-level traffic analysis pipeline. On a link of 1–10 Gbps, we might have, say, 10^4 concurrent connections, and thus we can then parallelize and/or

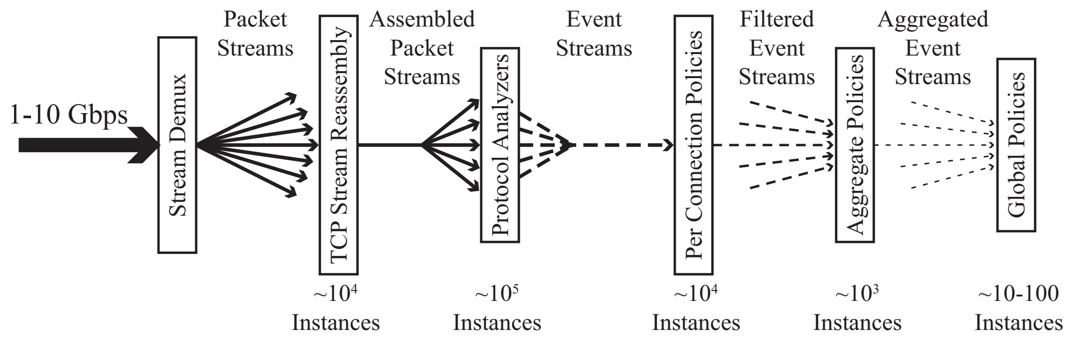


Figure 7.1: The spectrum of parallelism present in a high-level network security analysis pipeline.

pipeline the process of TCP stream reassembly and normalization amongst these 10^4 independent streams. At the protocol analysis stage, the amount of parallelism is further amplified because one needs to run different possible application parsers in parallel to dynamically determine which parser finds the flow syntactically and semantically consistent. "Events"—the outcome of protocol analysis—reflect a distillation of application-layer activities and can be analyzed first on a per-connection basis, thus this stage maintains the earlier parallelism at transport layer. Finally the system aggregates selected information across connections, sessions, and hosts, where the number of potential parallelizable tasks range from 10 to 1,000.

While there is more than sufficient amount of potential parallelism to explore in traffic analysis, the major challenges of exposing parallelism and mapping the pipeline to hardware remain. We envision pursuing these using three fundamental elements: (1) a high-level language for expressing rich forms of network analysis; (2) a powerful abstraction of parallel execution to which we target compilation of the elements of the language; and (3) a final step of compilation from that abstraction to concrete hardware implementations. The `binpac` language presented in this dissertation offers an important first step towards (1), as protocol parsing is one of the most cycle-demanding tasks in the pipeline. A high-level specification of the parsers in `binpac` provides a complete picture of data dependency inside and between connections—the key information the compiler needs for parallelization.

Learning protocol syntax from examples. Currently we build protocol parsers according to the syntax specification in protocol standards. However, there are many situations in which it is very useful to infer protocol syntax from traffic examples. First, it would allow us to analyze protocols without publicly available specification. A number of important network applications, for example, Microsoft SQL database server and Kazaa file sharing system, use *proprietary*, instead of open, protocols. Similarly, it also helps when the protocol specification is incomplete or outdated. Second, protocol learning can also expedite the process of specifying a complex protocol, such as the complete set of DCE/RPC services on Windows systems, by generating the protocol parsers (e.g., in the `binpac` language). As an intermediate goal, we envision semi-automated protocol learning where human and learning algorithms interactively discover protocol syntax and generate protocol parsers.

A first step towards protocol learning is represented by the RolePlayer traffic replay system, by Cui et al. [22]. RolePlayer tries to infer the protocol syntax by looking for some common elements (length field, cookie, IP addresses, and ports) based on contextual information—observing which elements vary with the environment and which do not—and by aligning and contrasting multiple samples of similar protocol conversation. The initial results are encouraging—RolePlayer is able to learn and replay complex conversation in protocols such as CIFS and FTP. Yet much more remains for exploration, in terms of being able to leverage more traffic samples, handling more protocols, and recognizing more complex syntax structures.

7.2.2 Traffic Anonymization

The anonymizations techniques proposed in this dissertation are an early push towards making richer packet traces available to the research community. There is still much to be done in this area. From our experience, we believe the main challenges include: (1) to formalize security risks and the process of developing an anonymization scheme; (2) to automate the process of anonymization and verification; (3) to keep more packet dynamics in the transformed traces. Below we briefly discuss each of these.

Formalizing anonymization. In Section 4.4 we describe our methodology for trace anonymization and analyzed four types of inference techniques, but our analysis is not formal or complete. While accumulation of experience will help us have a better understanding of the relationship among various data elements, developing a formal model for anonymization would be a big step forward beyond the intuitive methods. A formal model would mean that users can have a complete view of the threats and rigorously deduce a detailed anonymization scheme from the objectives. However, a fundamental difficulty in pursuing such a model is the very rich semantics network traces encompass—such semantics are difficult to capture with concise high-level abstractions and often involve corner cases that can inadvertently leak information.

Automating the anonymization process. Although the anonymization process has been much simplified by operating at the application-protocol level, currently we still need human assistance in tailoring scripts for traces (4.5.3), processing free-format texts (4.5.4), and result verification (4.5.5). The first two, though being optional, often largely improve the quality of the output trace. The last one—verification—is an essential step which we currently cannot perform without human interaction. On the other hand, fully automating anonymization will bring substantial benefits: (1) it will minimize human effort in releasing traces, making it easier for sites to make traces available; (2) it is essential for environments where the trace providers themselves are not allowed to see the original traffic (e.g., for traces collected at some ISPs); (3) automated verification will foster a model of "script↔data" exchange, where users send anonymization scripts to data owners who use them to easily generate traces returned to the users [67].

The key for automating result verification is to make the anonymization scheme "understandable" to the verifier program. One way is to design a declarative (instead of procedural) language for the anonymization scripts. Being declarative, the anonymization scheme specification is also amenable to verification, which is necessary to ensure that the scheme is correctly specified.

Keeping traffic dynamics. One fundamental difficulty of keeping the original traffic dynamics is that lengths of data may be changed during transformation, and the new lengths must be reflected in TCP/IP headers to keep packets "well-formed". Therefore there is not a single best way to keep the original dynamics. One possibility is to create an out-of-band channel to convey information such as original packet lengths, fragmentation, retransmission, etc.

Also, because traffic parsing is stateful, it is difficult to process two parallel versions of the data—for instance, when there are inconsistent TCP retransmissions. So we have to remove at least one version from the anonymized stream, even though in some contexts (e.g., analyzing possible intrusion detection evasions seen in practice [89]) it would be very useful to have both copies of inconsistent retransmissions retained.

7.3 Summary

This dissertation explores *semantic traffic analysis*—examining application-level data of network traffic to understand the behavior of network applications. The dissertation first describes `binpac`—a declarative language for building protocol analyzers, and tools and techniques for creating shared traffic data through anonymization of packet traces. We then present characterizations of two types of previously unstudied traffic—Internet background radiation and enterprise internal network traffic—with a focus on application semantics, as examples of what one can learn about network applications through semantic traffic analysis.

Appendix: A Sample HTTP Trace Transformation

The original trace was collected by `tcpdump` recording a retrieval of the www.google.com homepage. The `tcpdump` output (with wrapped packet summary lines and TCP payloads) of the original trace is shown on the next page.

We use our tool to transform the trace with a script that:

1. Replaces the data entity with its MD5 hash value (in this case, "`867119294265e3f445708c3fcfb2144f`");
2. Rewrites the `Content-length` field to reflect the length of the MD5 hash value;
3. Adds the header: "X-Actual-Data-Length: 2709; gap=0, content-length= 2709" to record the original Content-length field and how many bytes are actually transferred.

The `tcpdump` output of the transformed trace is also on the next page.

Note that "Write-Deferring" is applied here: the new headers are written at the position of the original `Content-length` header, even though the actual data size is not determined until all of the data is seen. The script defers writing the headers until the end of the message and then writes back to the reserved position.

Furthermore, by changing only one line of the script, from:

```
msg$abstract = md5_hash(data);
```

to:

```
    msg$abstract =
     subst_string(data, "Google", "Gooooogle");
```

the script then replaces every occurrence of “Google” in the data entity with “Gooooogle”, instead of replacing the whole data entity with its MD5 hash value. Next page shows part of the transformed trace. (There are four occurrences of “Google” in the original message, thus the Content-length increases from 2709 to 2717.) Note that sequence and acknowledgment numbers between the traces differ due to packet reframing and the addition of X-Actual-Data-Length headers.

Original trace:

```
1044328495.549695 192.150.187.28.1472 > 216.239.51.101.80:
    S 1352447574:1352447574(0) win 57344
    <mss 1460,nop,wscale 0,nop,nop,timestamp 92919815 0> (DF)
1044328495.632608 216.239.51.101.80 > 192.150.187.28.1472:
    S 3009119707:3009119707(0) ack 1352447575 win 1460
    <mss 1460,nop,nop,timestamp 752104543 92919815,nop,wscale 0> (DF)
1044328495.632647 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 1 win 57920
    <nop,nop,timestamp 92919823 752104543> (DF)
1044328495.632966 192.150.187.28.1472 > 216.239.51.101.80:
    P 1:81(80) ack 1 win 57920
    <nop,nop,timestamp 92919823 752104543> (DF)
0x0030  2cd4 345f 4745 5420 2f20 4854 5450 2f31 ,.4_GET./.HTTP/1
0x0040  2e30 0d0a 5573 6572 2d41 6765 6e74 3a20 .0..User-Agent:.
0x0050  5767 6574 2f31 2e35 2e33 0d0a 486f 7374 Wget/1.5.3..Host
0x0060  3a20 7777 772e 676f 6f67 6c65 2e63 6f6d :.www.google.com
0x0070  3a38 300d 0a41 6363 6570 743a 202a 2f2a :80..Accept:.*/*
0x0080  0d0a 0d0a                               ....
1044328495.716691 216.239.51.101.80 > 192.150.187.28.1472:
    . ack 81 win 30660
    <nop,nop,timestamp 752104551 92919823> (DF)
1044328495.737787 216.239.51.101.80 > 192.150.187.28.1472:
    P 1:1449(1448) ack 81 win 31856
    <nop,nop,timestamp 752104553 92919823> (DF)
0x0030  0589 d80f 4854 5450 2f31 2e30 2032 3030 ....HTTP/1.0.200
0x0040  204f 4b0d 0a43 6f6e 7465 6e74 2d4c 656e .OK..Content-Len
0x0050  6774 683a 2032 3730 390d 0a43 6f6e 6e65 gth:.2709..Conne
0x0060  6374 696f 6e3a 2043 6c6f 7365 0d0a 5365 ction:.Close..Se
0x0070  7276 6572 3a20 4757 532f 322e 300d 0a44 rver:.GWS/2.0..D
0x0080  6174 653a 2054 7565 2c20 3034 2046 6562 ate:.Tue,.04.Feb
0x0090  2032 3030 3320 3033 3a31 343a 3535 2047 .2003.03:14:55.G
0x00a0  4d54 0d0a 436f 6e74 656e 742d 5479 7065 MT..Content-Type
0x00b0  3a20 7465 7874 2f68 746d 6c0d 0a43 6163 :.text/html..Cac
0x00c0  6865 2d63 6f6e 7472 6f6c 3a20 7072 6976 he-control:.priv
0x00d0  6174 650d 0a53 6574 2d43 6f6f 6b69 653a ate..Set-Cookie:
0x00e0  2050 5245 463d 4944 3d31 6538 6337 3538 .PREF=ID=1e8c758
0x00f0  6231 6632 3965 3836 643a 544d 3d31 3034 b1f29e86d:TM=104
0x0100  3433 3238 3439 353a 4c4d 3d31 3034 3433 4328495:LM=10443
0x0110  3238 3439 353a 533d 6638 344d 6753 7948 28495:S=f84MgSyH
0x0120  3347 452d 3439 5070 3b20 6578 7069 7265 3GE-49Pp;.expire
0x0130  733d 5375 6e2c 2031 372d 4a61 6e2d 3230 s=Sun,.17-Jan-20
0x0140  3338 2031 393a 3134 3a30 3720 474d 543b 38.19:14:07.GMT;
0x0150  2070 6174 683d 2f3b 2064 6f6d 6169 6e3d .path=/;.domain=
0x0160  2e67 6f6f 676c 652e 636f 6d0d 0a0d 0a3c .google.com....<
0x0170  6874 6d6c 3e3c 6865 6164 3e3c 6d65 7461 html><head><meta
0x0180  2068 7474 702d 6571 7569 763d 2263 6f6e .http-equiv="con
0x0190  7465 6e74 2d74 7970 6522 2063 6f6e 7465 tent-type".conte
0x01a0  6e74 3d22 7465 7874 2f68 746d 6c3b 2063 nt="text/html;.c
0x01b0  6861 7273 6574 3d49 534f 2d38 3835 392d harset=ISO-8859-
0x01c0  3122 3e3c 7469 746c 653e 476f 6f67 6c65 1"><title>Google
0x01d0  3c2f 7469 746c 653e 3c73 7479 6c65 3e3c </title><style><
...
0x0360  3237 3620 6865 6967 6874 3d31 3130 2061 276.height=110.a
0x0370  6c74 3d22 476f 6f67 6c65 223e 3c2f 7464 lt="Google"></td
...
1044328495.737951 216.239.51.101.80 > 192.150.187.28.1472:
    P 2897:3025(128) ack 81 win 31856
    <nop,nop,timestamp 752104553 92919823> (DF)
0x0030  0589 d80f 6f6e 743e 0a3c 703e 3c66 6f6e ....ont>.<p><fon
0x0040  7420 7369 7a65 3d2d 323e 2663 6f70 793b t.size=-2>&copy;
0x0050  3230 3033 2047 6f6f 676c 653c 2f66 6f6e 2003.Google</fon
0x0060  743e 3c66 6f6e 7420 7369 7a65 3d2d 323e t><font.size=-2>
0x0070  202d 2053 6561 7263 6869 6e67 2033 2c30 .-.Searching.3,0
...
1044328495.737987 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 1449 win 57920
    <nop,nop,timestamp 92919833 752104553> (DF)
1044328495.738022 216.239.51.101.80 > 192.150.187.28.1472:
    F 3025:3025(0) ack 81 win 31856
    <nop,nop,timestamp 752104553 92919823> (DF)
1044328495.738054 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 1449 win 57920
    <nop,nop,timestamp 92919833 752104553> (DF)
1044328495.739267 216.239.51.101.80 > 192.150.187.28.1472:
    P 1449:2897(1448) ack 81 win 31856
    <nop,nop,timestamp 752104553 92919823> (DF)
0x0030  0589 d80f 2f66 6f6e 743e 3c2f 613e 3c2f ..../font></a></
0x0040  7464 3e3c 7464 2077 6964 7468 3d31 353e td><td.width=15>
0x0050  266e 6273 703b 3c2f 7464 3e3c 7464 2069 &nbsp;</td><td.i
0x0060  643d 3320 6267 636f 6c6f 723d 2365 6665 d=3.bgcolor=#efe
0x0070  6665 6620 616c 6967 6e3d 6365 6e74 6572 fef.align=center
...
0x0370  7562 6d69 7420 7661 6c75 653d 2247 6f6f ubmit.value="Goo
0x0380  676c 6520 5365 6172 6368 2220 6e61 6d65 gle.Search".name
...
1044328495.739318 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 3026 win 56344
    <nop,nop,timestamp 92919833 752104553> (DF)
1044328495.741006 192.150.187.28.1472 > 216.239.51.101.80:
    F 81:81(0) ack 3026 win 57920
    <nop,nop,timestamp 92919834 752104553> (DF)
1044328495.823516 216.239.51.101.80 > 192.150.187.28.1472:
    . ack 82 win 31856
    <nop,nop,timestamp 752104562 92919834> (DF)
```

Replacing data entity with MD5 hash value:

```
1044328495.549695 192.150.187.28.1472 > 216.239.51.101.80:
    S 1352447574:1352447574(0) win 57344
    <mss 1460,nop,wscale 0,nop,nop,timestamp 92919815 0>
1044328495.632608 216.239.51.101.80 > 192.150.187.28.1472:
    S 3009119707:3009119707(0) ack 1352447575 win 1460
    <mss 1460,nop,nop,timestamp 752104543 92919815,nop,wscale 0>
1044328495.632647 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 1 win 57920
    <nop,nop,timestamp 92919823 752104543>
1044328495.632966 192.150.187.28.1472 > 216.239.51.101.80:
    P 1:130(129) ack 1 win 57920
    <nop,nop,timestamp 92919823 752104543>
0x0030  2cd4 345f 4745 5420 2f20 4854 5450 2f31 ,.4_GET./.HTTP/1
0x0040  2e30 0d0a 5553 4552 2d41 4745 4e54 3a20 .0..USER-AGENT:.
0x0050  5767 6574 2f31 2e35 2e33 0d0a 484f 5354 Wget/1.5.3..HOST
0x0060  3a20 7777 772e 676f 6f67 6c65 2e63 6f6d :.www.google.com
0x0070  3a38 300d 0a41 4343 4550 543a 202a 2f2a :80..ACCEPT:.*/*
0x0080  0d0a 0d0a 582d 4163 7475 616c 2d44 6174 ....X-Actual-Dat
0x0090  612d 4c65 6e67 7468 3a20 303b 2067 6170 a-Length:.0;.gap
0x00a0  3d30 2c20 636f 6e74 656e 742d 6c65 6e67 =0,.content-leng
0x00b0  7468 3d0d 0a                            th=..
1044328495.716691 216.239.51.101.80 > 192.150.187.28.1472:
    . ack 130 win 30660
    <nop,nop,timestamp 752104551 92919823>
1044328495.737787 216.239.51.101.80 > 192.150.187.28.1472:
    P 1:371(370) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
0x0030  0589 d80f 4854 5450 2f31 2e30 2032 3030 ....HTTP/1.0.200
0x0040  204f 4b0d 0a43 6f6e 7465 6e74 2d4c 656e .OK..Content-Len
0x0050  6774 683a 2033 320d 0a58 2d41 6374 7561 gth:.32..X-Actua
0x0060  6c2d 4461 7461 2d4c 656e 6774 683a 2032 l-Data-Length:.2
0x0070  3730 393b 2067 6170 3d30 2c20 636f 6e74 709;.gap=0,.cont
0x0080  656e 742d 6c65 6e67 7468 3d20 3237 3039 ent-length=.2709
0x0090  0d0a 434f 4e4e 4543 5449 4f4e 3a20 436c ..CONNECTION:.Cl
0x00a0  6f73 650d 0a53 4552 5645 523a 2047 5753 ose..SERVER:.GWS
0x00b0  2f32 2e30 0d0a 4441 5445 3a20 5475 652c /2.0..DATE:.Tue,
0x00c0  2030 3420 4665 6220 3230 3033 2030 333a .04.Feb.2003.03:
0x00d0  3134 3a35 3520 474d 540d 0a43 4f4e 5445 14:55.GMT..CONTE
0x00e0  4e54 2d54 5950 453a 2074 6578 742f 6874 NT-TYPE:.text/ht
0x00f0  6d6c 0d0a 4341 4348 452d 434f 4e54 524f ml..CACHE-CONTRO
0x0100  4c3a 2070 7269 7661 7465 0d0a 5345 542d L:.private..SET-
0x0110  434f 4f4b 4945 3a20 5052 4546 3d49 443d COOKIE:.PREF=ID=
0x0120  3165 3863 3735 3862 3166 3239 6538 3664 1e8c758b1f29e86d
0x0130  3a54 4d3d 3130 3434 3332 3834 3935 3a4c :TM=1044328495:L
0x0140  4d3d 3130 3434 3332 3834 3935 3a53 3d66 M=1044328495:S=f
0x0150  3834 4d67 5379 4833 4745 2d34 3950 703b 84MgSyH3GE-49Pp;
0x0160  2065 7870 6972 6573 3d53 756e 2c20 3137 .expires=Sun,.17
0x0170  2d4a 616e 2d32 3033 3820 3139 3a31 343a -Jan-2038.19:14:
0x0180  3037 2047 4d54 3b20 7061 7468 3d2f 3b20 07.GMT;.path=/;.
0x0190  646f 6d61 696e 3d2e 676f 6f67 6c65 2e63 domain=.google.c
0x01a0  6f6d 0d0a 0d0a                          om....
1044328495.737987 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 371 win 57920
    <nop,nop,timestamp 92919833 752104553>
1044328495.739267 216.239.51.101.80 > 192.150.187.28.1472:
    FP 371:403(32) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
0x0030  0589 d80f 3836 3731 3139 3239 3432 3635 ....867119294265
0x0040  6533 6634 3435 3730 3863 3366 6366 6232 e3f445708c3fcfb2
0x0050  3134 3466                               144f
1044328495.739318 192.150.187.28.1472 > 216.239.51.101.80:
    . ack 404 win 56344
    <nop,nop,timestamp 92919833 752104553>
1044328495.741006 192.150.187.28.1472 > 216.239.51.101.80:
    F 130:130(0) ack 404 win 57920
    <nop,nop,timestamp 92919834 752104553>
1044328495.823516 216.239.51.101.80 > 192.150.187.28.1472:
    . ack 131 win 31856
    <nop,nop,timestamp 752104562 92919834>
```

Substituting “Google” with “Goooogle”:

```
1044328495.737787 216.239.51.101.80 > 192.150.187.28.1472:
    P 1:373(372) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
0x0030  0589 d80f 4854 5450 2f31 2e30 2032 3030 ....HTTP/1.0.200
0x0040  204f 4b0d 0a43 6f6e 7465 6e74 2d4c 656e .OK..Content-Len
0x0050  6774 683a 2032 3731 370d 0a58 2d41 6374 gth:.2717..X-Act
0x0060  7561 6c2d 4461 7461 2d4c 656e 6774 683a ual-Data-Length:
0x0070  2032 3730 393b 2067 6170 3d30 2c20 636f .2709;.gap=0,.co
0x0080  6e74 656e 742d 6c65 6e67 7468 3d20 3237 ntent-length=.27
0x0090  3039 0d0a 434f 4e4e 4543 5449 4f4e 3a20 09..CONNECTION:.
...
1044328495.739267 216.239.51.101.80 > 192.150.187.28.1472:
    P 373:1821(1448) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
...
0x0080  3838 3539 2d31 223e 3c74 6974 6c65 3e47 8859-1"><title>G
0x0090  6f6f 6f6f 676c 653c 2f74 6974 6c65 3e3c ooogle</title><
...
0x0230  743d 3131 3020 616c 743d 2247 6f6f 6f6f t=110.alt="Goooo
0x0240  676c 6522 3e3c 2f74 643e 3c2f 7472 3e3c gle"></td></tr><
...
1044328495.739267 216.239.51.101.80 > 192.150.187.28.1472:
    F 1821:3090(1269) ack 130 win 31856
    <nop,nop,timestamp 752104553 92919823>
...
0x0230  7574 2074 7970 653d 7375 626d 6974 2076 ut.type=submit.v
0x0240  616c 7565 3d22 476f 6f6f 6f67 6c65 2053 alue="Goooogle.S
0x0250  6561 7263 6822 206e 616d 653d 6274 6e47 earch".name=btnG
...
0x04c0  7079 3b32 3030 3320 476f 6f6f 6f67 6c65 py;2003.Goooogle
0x04d0  3c2f 666f 6e74 3e3c 666f 6e74 2073 697a </font><font.siz
...
```

Bibliography

[1] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, 1993.

[2] W32 Agobot IB. http://www.sophos.com/virusinfo/analyses/trojagobotib.html.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] William Aiello, Chuck Kalmanek, Patrick McDaniel, Shubo Sen, Oliver Spatscheck, and K. van der Merwe. Analysis of communities of interest in data networks. In *Proceedings of Passive and Active Measurement Workshop (PAM)*, March 2005.

[5] M. Allman, E. Blanron, and W. Eddy. A scalable system for sharing Internet measurement. In *Proceedings of Passive and Active Measurement Workshop (PAM)*, 2002.

[6] M. Arlitt and C. Williamson. Web server workload characterization: The search for invariants. In *Proceedings of ACM SIGMETRICS*, Philadelphia, PA, May 1996.

[7] Martin Arlitt, Balachander Krishnamurthy, and Jeffrey C. Mogul. Predicting short-transfer latency from TCP arcana: A trace-based validation. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2005.

[8] *Abstract Syntax Notation One (ASN.1)*. ISO/IEC 8824-1:2002.

[9] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security*, 3(3):186–205, August 2000.

[10] Godmar Back. DataScript—a specification and scripting language for binary data. In *GPCE '02: The ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, pages 66–77, London, UK, 2002. Springer-Verlag.

[11] Lawrence Baldwin, Philip Sloss, and Steve Friedl. Iraqi trace. http://www.mynetwatchman.com/kb/security/articles/iraqiworm/iraqitrace.htm.

[12] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of ACM SIGMETRICS*, pages 151–160, July 1998.

[13] W32 Beagle.J. http://securityresponse.symantec.com/avcenter/venc/data/w32.beagle.j@mm.html.

[14] Edoardo Biagioni, Robert Harper, and Peter Lee. A network protocol stack in Standard ML. *Higher-Order and Symbolic Computation*, 14(4):309–356, 2001.

[15] Nikita Borisov, David J. Brumley, Helen J. Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo. Generic application-level protocol analyzer and its language. Under submission.

[16] Ramon Cáceres. Measurements of wide area Internet traffic. Technical report, EECS, University of California, Berkeley, 1989.

[17] Ramon Caceres, Peter Danzig, Sugih Jamin, and Danny Mitzel. Characteristics of wide-area TCP/IP conversations. In *Proceedings of the ACM SIGCOMM Conference*, 1991.

[18] Capture the capture the flag. http://www.shmoo.com/cctf/.

[19] Common Internet File System. http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf.

[20] D. Crocker. *RFC 2234: Augmented BNF for Syntax Specifications: ABNF*.

[21] Weidong Cui, Vern Paxson, and Nicholas Weaver. Gq: Realizing a system to catch worms in a quarter million places. Under submission.

[22] Weidong Cui, Vern Paxson, Nicholas Weaver, and Randy H. Katz. Protocol-independent adaptive replay of application dialog. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2006.

[23] M. Dacier, F. Pouget, and H. Debar. Attack processes found on the Internet. In *Proceedings of NATO Symposium*, 2004.

[24] Peter Danzig, S. Jamin, R. Cáceres, D. Mitzel, and D. Estrin. An empirical workload model for driving wide-area TCP/IP network simulations. *Internetworking: Research and Experience*, 3(1):1–26, 1992.

[25] DCE 1.1: Remote procedure call. http://www.opengroup.org/onlinepubs/9629399/toc.htm.

[26] H. Dreger, A. Feldmann, M. Mai, V. Paxson, and R. Sommer. Dynamic application-layer protocol analysis for network intrusion detection. In *Proceedings of USENIX Security Symposium*, August 2006.

[27] DSniff. www.monkey.org/ dugsong/dsniff.

[28] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of Email and research workloads. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[29] *The Ethereal Network Analyzer*. http://www.ethereal.com/.

[30] Federal Committee on Statistical Methodology. Report on statistical disclosure limitation methodology (statistical policy working paper 22), 1994. http://www.fcsm.gov/working-papers/spwp22.html.

[31] Anja Feldmann. BLT: Bi-layer tracing of HTTP and TCP/IP. In *Proceedings of World Wide Web Conference*, May 2000.

[32] Anja Feldmann, Nils Kammenhuber, Olaf Maennel, Bruce Maggs, Roberto De Prisco, and Ravi Sundaram. A methodology for estimating interdomain web traffic demand. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2004.

[33] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, June 1999.

[34] Kathleen Fisher and Robert Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 295–304, New York, NY, USA, 2005. ACM Press.

[35] Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. The next 700 data description languages. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 2–15, New York, NY, USA, 2006. ACM Press.

[36] Flowreplay design notes. http://www.synfin.net/papers/flowreplay.pdf.

[37] Sally Floyd, Mark Allman, A. Jain, and Pasi Sarolahti. Quick-start for TCP and IP, June 2006. Internet-Draft draft-ietf-tsvwg-quickstart-04.txt (work in progress).

[38] H. J. Fowler and W. E. Leland. Local area network traffic characteristics, with implications for broadband network congestion management. *IEEE Journal on Selected Areas in Communications*, SAC-9:1139–49, 1991.

[39] Luiz Gomes, Cristiano Cazita, Jussara Almeida, Virgilio Almeida, and Wagner Meira Jr. Characterizing a SPAM traffic. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2004.

[40] B. Greene. BGPv4 Security Risk Assessment, June 2002.

[41] Steven D. Gribble and Eric A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, December 1997.

[42] R. Gusella. A measurement study of diskless workstation traffic on an Ethernet. *IEEE Transactions on Communications*, 38(9):1557–1568, September 1990.

[43] Mark Handley, Christian Kreibich, and Vern Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of USENIX Security Symposium*, 2001.

[44] The Honeynet Project. http://project.honeynet.org, 2003.

[45] The honeypot challenge. http://project.honeynet.org/misc/chall.html.

[46] The IANA port assignment. http://www.iana.org/assignments/port-numbers.

[47] S. Ita. s_ita's port list. http://www.bekkoame.ne.jp/∼s_ita/port/.

[48] Editor J. Klensin. *RFC 2821: Simple Mail Transfer Protocol*, April 2001.

[49] Van Jacobson, Craig Leres, and Steven McCanne. *TCPDUMP*. ftp://ftp.ee.lbl.gov/libpcap.tar.Z.

[50] Xuxian Jiang and Dongyan Xu. Collapsar: A VM-based architecture for network attack detention center. In *Proceedings of USENIX Security Symposium*, San Diego, CA, August 2004.

[51] Jaeyeon Jung, Vern Paxson, Arthur Berger, and Hari Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[52] Jaeyeon Jung and Emil Sit. An empirical study of spam traffic and the use of DNS black lists. In *Proceedings of the Internet Measurement Conference (IMC)*, Taormina, Italy, October 2004.

[53] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. DNS performance and the effectiveness of caching. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2001.

[54] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of USENIX Security Symposium*, San Diego, CA, August 2004.

[55] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[56] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. In *Proceedings of the ACM SIGCOMM Conference*, pages 3–13, Cambridge, MA, August 1999.

[57] Tadayoshi Kohno, Andre Broido, and kc claffy. Remote physical device fingerprinting. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[58] Christian Kreibich. *NetDuDe (NETwork DUmp data Displayer and Editor)*. . http://netdude.sourceforge.net/.

[59] Christian Kreibich and John Crowcroft. Honeycomb–creating intrusion detection signatures using honeypots. In *Proceedings of Workshop on Hot Topics in Networks (HotNets)*, Cambridge, MA, November 2003.

[60] Abhishek Kumar, Vern Paxson, and Nicholas Weaver. Exploiting underlying structure for detailed reconstruction of an Internet-scale event. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2005.

[61] Richard Lippmann, Seth Webster, and Douglas Stetson. The effect of identifying vulnerabilities and patching software on the utility of network intrusion detection. In *Proceedings of Recent Advances in Intrusion Detection*, number 2516 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[62] M. Lottor. *RFC 1296: Internet Growth (1981-1991)*.

[63] Bruce Mah. An empirical model of HTTP network traffic. In *Proceedings of IEEE INFOCOM*, April 1997.

[64] G. Robert Malan and Farnam Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of the ACM SIGCOMM Conference*, 1998.

[65] Peter J. McCann and Satish Chandra. Packet Types: Abstract specifications of network protocol messages. In *Proceedings of the ACM SIGCOMM Conference*, pages 321–333, 2000.

[66] Greg Minshall. *TCPdpriv: Program for Eliminating Confidential Information from Traces*. Ipsilon Networks, Inc. http://ita.ee.lbl.gov/html/contrib/tcpdpriv.html.

[67] Jeffrey Mogul. Trace anonymization misses the point. Presentation on WWW 2002 Panel on Web Measurements.

[68] D. Moore. Network telescopes: Observing small or distant security events. Invited Presentation at the 11th USENIX Security Symposium, 2002.

[69] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer worm. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, June 2003.

[70] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm. http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html, 2003.

[71] D. Moore, C. Shannon, and J. Brown. Code Red: A case study on the spread and victims of an Internet worm. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2002.

[72] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for containing self-propagating code. In *Proceedings of IEEE INFOCOM*, April 2003.

[73] D. Moore, G. Voelker, and S. Savage. Inferring Internet denial of service activity. In *Proceedings of USENIX Security Symposium*, Washington D.C., August 2001.

[74] W32 Mydoom.A@mm. . http://securityresponse.symantec.com/avcenter/venc/data/w32.mydoom.a@mm.html.

[75] *NFR Security*. http://www.nfr.com.

[76] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of the ACM SIGCOMM Conference*, September 1998.

[77] Ruoming Pang, Mark Allman, Mike Bennett, Jason Lee, Vern Paxson, and Brian Tierney. A first look at modern enterprise traffic. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2005.

[78] Ruoming Pang, Mark Allman, Vern Paxson, and Jason Lee. The devil and packet trace anonymization. *SIGCOMM Computer Communication Review*, 36(1):29–38, 2006.

[79] Ruoming Pang and Vern Paxson. Anonymized FTP traces. http://www-nrg.ee.lbl.gov/anonymized-traces.html.

[80] Ruoming Pang and Vern Paxson. A high-level programming environment for packet trace anonymization and transformation. In *Proceedings of the ACM SIGCOMM Conference*, August 2003.

[81] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet background radiation. In *Proceedings of the Internet Measurement Conference (IMC)*, October 2004.

[82] LBNL/ICSI enterprise tracing project. http://www.icir.org/enterprise-tracing/.

[83] T. Parr and R. Quong. ANTLR: A predicated-LL (k) parser generator. *Software, Practice and Experience*, 25, July 1995.

[84] Simon Patarin and Mesaac Makpangou. Pandora: A flexible network monitoring platform. In *Proceedings of the USENIX Annual Technical Conference*, San Diego, June 2000.

[85] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking hardware support for network analysis and intrusion prevention. In *Proceedings of Workshop on Hot Topics in Security (HotSec)*, Vancouver, B.C., Canada, July 2006.

[86] Vern Paxson. *Bro: A System for Detecting Network Intruders in Real-Time*. http://www.bro-ids.org.

[87] Vern Paxson. Empirically-derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, August 1994.

[88] Vern Paxson. Growth trends in wide-area TCP connections. *IEEE Network*, 8(4):8–17, July/August 1994.

[89] Vern Paxson. Bro: A system for detecting network intruders in real time. *Computer Networks*, December 1999.

[90] Vern Paxson. Strategies for sound Internet measurement. In *Proceedings of the Internet Measurement Conference (IMC)*, 2004.

[91] Markus Peuhkuri. A method to compress and anonymize packet traces. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2001.

[92] Jon Postel. *RFC 768: User Datagram Protocol*, August 1980.

[93] Jon Postel. *RFC 791: Internet Protocol*, September 1981.

[94] Jon Postel. *RFC 793: Transmission Control Protocol*, September 1981.

[95] *NetWare Core Protocol*. http://forge.novell.com/modules/xfmod/project/?ncp.

[96] Niels Provos. The Honeyd Virtual Honeypot. http://www.honeyd.org.

[97] Niels Provos. A virtual honeypot framework. In *Proceedings of USENIX Security Symposium*, San Diego, CA, August 2004.

[98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[99] W32 Randex.D. http://www.liutilities.com/products/wintaskspro/processlibrary/msmsgri32.

[100] M. Roesch. SNORT: Lightweight intrusion detection for networks. In *Proceedings of USENIX LISA*, 1999.

[101] The SNORT network intrusion detection system. http://www.snort.org.

[102] Stefan Saroiu, Krishna P. Gummadi, Richard J. Dunn, Steven D. Gribble, and Henry M. Levy. An analysis of Internet content delivery systems. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[103] W32 Sasser.Worm. . http://securityresponse.symantec.com/avcenter/venc/data/w32.sasser.worm.html.

[104] Stefen Savage. Private communication.

[105] Security Focus. Microsoft IIS 5.0 "translate: f" source disclosure vulnerability. . http://www.securityfocus.com/bid/1578/discussion/, April 2004.

[106] Subhabrata Sen and Jia Wang. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop (IMW)*, pages 137–150, November 2002.

[107] Aman Shaikh, Chris Isett, Albert Greenberg, Matthew Roughan, and Joel Gottlieb. A case study of OSPF behavior in a large enterprise network. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop (IMW)*, pages 217–230, New York, NY, USA, 2002. ACM Press.

[108] Colleen Shannon and David Moore. The spread of the Witty worm. . http://www.caida.org/analysis/security/witty, 2004.

[109] S. Singh, C. Estan, G. Varghese, and S. Savage. The Earlybird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California, San Diego, August 2003.

[110] R. Srinivasan. *RFC 1831: RPC: Remote Procedure Call Protocol Specification*.

[111] R. Srinivasan. *RFC 1832: XDR: External Data Representation Standard*.

[112] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proceedings of USENIX Security Symposium*, 2002.

[113] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *Proceedings of the ACM SIGCOMM Conference*, September 2000.

[114] Q. Sun, D. R. Simon, Y. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu. Statistical identification of encrypted web browsing traffic. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[115] Godfrey Tan, Massimiliano Poletto, John Guttag, and Frans Kaashoek. Role classification of hosts within enterprise networks based on connection patterns. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[116] K. Thompson, G. Miller, and R. Wilder. Wide area Internet traffic patterns and characteristics. *IEEE Network*, 11(6):10–23, November 1997.

[117] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *RFC 2165: Service Location Protocol*.

[118] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[119] Ethereal OSPF protocol dissector buffer overflow vulnerability. . http://www.idefense.com/intelligence/vulnerabilities/display.php?id=349.

[120] Snort TCP stream reassembly integer overflow exploit. . http://www.securiteam.com/exploits/5BP0O209PS.html.

[121] Symantec multiple firewall NBNS response processing stack overflow. . http://www.eeye.com/html/research/advisories/AD20040512A.html.

[122] tcpdump ISAKMP packet delete payload buffer overflow. http://xforce.iss.net/xforce/xfdb/15680.

[123] DameWare Mini Remote Control Server <= 3.72 buffer overflow. . http://www.securityfocus.com/archive/1/347576.

[124] Microsoft Windows DCOM RPC interface buffer overrun vulnerability (MS03-026). . http://www.securityfocus.com/bid/8205.

[125] Microsoft Windows Locator Service buffer overflow vulnerability (MS03-001). . http://www.securityfocus.com/bid/6666.

[126] Microsoft Windows 2000 WebDAV buffer overflow vulnerability (MS03-007). . http://www.securityfocus.com/bid/7116.

[127] Separation of concerns. http://en.wikipedia.org/wiki/Separation_of_concerns.

[128] WildPackets, Inc. *EtherPeek*. http://www.etherpeek.com/.

[129] Windows Messenger popup spam. http://www.lurhq.com/popup_spam.html.

[130] Cynthia Wong, Stan Bielski, Jonathan M. McCune, and Chenxi Wang. A study of mass-mailing worms. In *Proceedings of the 2005 ACM Workshop on Rapid Malcode (WORM)*, pages 1–10, New York, NY, USA, 2004. ACM Press.

[131] W32 Xibo. http://www.sophos.com/virusinfo/analyses/w32xiboa.html.

[132] Jun Xu, Jinliang Fan, Mostafa Ammar, and Sue B. Moon. On the design and performance of prefix preserving IP traffic trace anonymization. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2001.

[133] V. Yegneswaran, P. Barford, and D. Plonka. On the design and use of Internet sinks for network abuse monitoring. In *Proceedings of Recent Advances in Intrusion Detection*, 2004.

[134] V. Yegneswaran, P. Barford, and J. Ullrich. Internet intrusions: Global characteristics and prevalence. In *Proceedings of ACM SIGMETRICS*, June 2003.

[135] T. Ylonen. Thoughts on how to mount an attack on TCPdpriv's "-a50" option. . http://ita.ee.lbl.gov/html/contrib/attack50/attack50.html.