Statistics for Engineering
and Information Science

Series Editors
M. Jordan, S.L. Lauritzen, J.F. Lawless, V. Nair

Springer Science+Business Media, LLC

Statistics for Engineering and Information Science

Akaike and Kitagawa: The Practice of Time Series Analysis.
Cowell, Dawid, Lauritzen, and Spiegelhalter: Probabilistic Networks and Expert Systems.
Doucet, de Freitas, and Gordon: Sequential Monte Carlo Methods in Practice.
Fine: Feedforward Neural Network Methodology.
Hawkins and Olwell: Cumulative Sum Charts and Charting for Quality Improvement.
Jensen: Bayesian Networks and Decision Graphs.
Marchette: Computer Intrusion Detection and Network Monitoring: A Statistical Viewpoint.
Vapnik: The Nature of Statistical Learning Theory, Second Edition.

David J. Marchette

Computer Intrusion Detection and Network Monitoring

A Statistical Viewpoint

With 86 Illustrations

Springer

David J. Marchette
Naval Surface Warfare Center
Code B10
17320 Dahlgren Road
Dahlgren, VA 22448
USA
marchettedj@nswc.navy.mil

Library of Congress Cataloging-in-Publication Data
Marchette, David J.
Computer intrusion detection and network monitoring : a statistical viewpoint / David J. Marchette.
p. cm. — (Statistics for engineering and information science)
Includes bibliographical references and index.
ISBN 978-1-4419-2937-2 ISBN 978-1-4757-3458-4 (eBook)
DOI 10.1007/978-1-4757-3458-4

1. Computer security—Statistical methods. 2. Computer networks—Security measures—Statistical methods. I. Title. II. Series.
QA76.9.A25 .M34 2001
005.8—dc21 2001032011

Printed on acid-free paper.

Originally published by Springer-Verlag New York, Inc. in 2001.

Softcover reprint of the hardcover 1st edition 2001

Production managed by Michael Koy; manufacturing supervised by Joe Quatela.
Photocomposed copy prepared from the author's LaTeX2e files.

9 8 7 6 5 4 3 2 1

SPIN 10833837

Preface

In the fall of 1999, I was asked to teach a course on computer intrusion detection for the Department of Mathematical Sciences of The Johns Hopkins University. That course was the genesis of this book. I had been working in the field for several years at the Naval Surface Warfare Center, in Dahlgren, Virginia, under the auspices of the SHADOW program, with some funding by the Office of Naval Research.

In designing the class, I was concerned both with giving an overview of the basic problems in computer security, and with providing information that was of interest to a department of mathematicians. Thus, the focus of the course was to be more on methods for modeling and detecting intrusions rather than one on how to secure one's computer against intrusions.

The first task was to find a book from which to teach. I was familiar with several books on the subject, but they were all at either a high level, focusing more on the political and policy aspects of the problem, or were written for security analysts, with little to interest a mathematician. I wanted to cover material that would appeal to the faculty members of the department, some of whom ended up sitting in on the course, as well as providing some interesting problems for students. None of the books on the market at the time had an adequate discussion of mathematical issues related to intrusion detection.

Lacking a text, I was thus forced to provide examples from articles, Web sites, and the like. After the course was over, I decided it would be a good idea to provide a compendium of the information that I had found. This book is the result. It's purpose is to provide an introduction to some of the issues in computer intrusion detection, with a focus on problems and techniques that would be of interest to a mathematician or statistician.

I have provided an extensive bibliography, covering much of the research in computer intrusion detection. This is not complete, but it does cover most of the important papers in the area.

My background is in pattern recognition and statistics, with a focus on computational statistics. This is the branch of statistics that is interested in the interface between statistics and computers. It considers issues related to computation, large data sets and high-dimensional data, visualization of complex data, and nonparametric models. Thus, computer intrusion detection was a natural area in which to become involved.

Dahlgren, Virginia, USA D.J. MARCHETTE

Acknowledgments

My mentors and teachers have had an important part in making this book possible. In particular, I am indebted to my dissertation advisor, Prof. Ed Wegman, for his encouragement, advice and friendship. I would not have learned about computer security and intrusion detection without John Green, Vicki Irwin and Stephen Northcutt. They have been extremely helpful, providing information, data, and training that were invaluable. I also want to thank the Department of Mathematical Sciences of The Johns Hopkins University, particularly Dan Naiman, John Wierman, Alan Goldman, and Carey Priebe. Many other people have had parts in bringing some of the information in this book to light, including Pat Carter, Jim Matthews, and Jeff Solka. Glen Moore, my boss at NSWC, has been extremely supportive, as has Wendy Martinez of the Office of Naval Research. I would particularly like to thank Matt Schonlau and Bill Cheswick for allowing me to use their graphics. Fred Kirby offered suggestions and caught several glaring errors. John Kimmel has been instrumental in bringing this work to fruition with a minimum of pain and suffering on my part. Finally, I must thank my family, particularly Susan, who has put up with a lot and spent a lot of time reading through and correcting manuscripts. The errors that remain are there in spite of her heroic efforts. Also, thanks to Steven, Jeffrey, and Katy for putting up with me while this work was written.

D.J. MARCHETTE

Contents

Preface *v*
Acknowledgments *vii*
Introduction *xiii*

Part I Networking Basics

1 TCP/IP Networking *3*
1.1 Overview of Networking 3
1.2 tcpdump 6
1.3 Network Layering 9
1.4 Data Encapsulation 10
1.5 Header Information 11
1.6 Fragmentation 21
1.7 Routing 22
1.8 Domain Name Service 23
1.9 Miscellaneous Utilities 27
1.10 Further Reading 42

2 Network Statistics *43*
2.1 Introduction 43
2.2 Network Traffic Intensities 43

2.3 Modeling Network Traffic 53
2.4 Mapping the Internet 58
2.5 Visualizing Network Traffic 60
2.6 Further Reading 70

3 Evaluation *73*
3.1 Introduction 73
3.2 Evaluating Classifiers 75
3.3 Receiver Operator Characteristic Curves 79
3.4 The DARPA/MITLL ID Testbed 79
3.5 Live Network Testing 82
3.6 Further Reading 84

Part II Intrusion Detection

4 Network Monitoring *89*
4.1 Introduction 89
4.2 tcpdump Filters 90
4.3 Common Attacks 91
4.4 SHADOW 106
4.5 Activity Profiling 109
4.6 EMERALD 146
4.7 WATCHERS 150
4.8 GrIDS 150
4.9 Miscellaneous Utilities 151
4.10 Further Reading 157

5 Host Monitoring *159*
5.1 Introduction 159
5.2 Common Attacks 159
5.3 NIDES 171
5.4 Computer Immunology 178
5.5 User Profiling 183
5.6 Miscellaneous Utilities 201
5.7 Further Reading 209

Part III Viruses and Other Creatures

6 Computer Viruses and Worms *215*
6.1 Introduction 215
6.2 How Viruses Replicate 216
6.3 How Viruses Scanners Work 218
6.4 Epidemiology 221
6.5 An Immunology Approach 229
6.6 Virus Phylogenies 231
6.7 Computer Worms 232
6.8 Further Reading 239

7 Trojan Programs and Covert Channels *241*
7.1 Introduction 241
7.2 Covert Channels 242
7.3 Steganography 246
7.4 Back Doors 249
7.5 Miscellaneous Trojans 252
7.6 Detecting Trojans 254
7.7 Further Reading 255

Appendix A Well-Known Port Numbers *257*

Appendix B Trojan Port Numbers *265*

Appendix C Country Codes *275*

Appendix D Security Web Sites *281*
D.1 Introduction 281
D.2 General Information Web Sites 282
D.3 Security 284
D.4 Cyber Crime 287
D.5 Software 288
D.6 Data 289
D.7 Intrusion Detection 289

Bibliography *291*

Glossary *311*

Acronyms *317*

Author Index *320*

Subject Index *325*

Introduction

Computer networks are a rich source of interesting problems and data for statisticians. This book will explore some of the issues of interest to the statistician that arise from the general problem of protecting computers and computer networks from unauthorized use or malicious attacks. This book will not attempt to be comprehensive, but rather will focus on a few areas of particular interest that lend themselves to statistical or probabilistic analysis.

One reason to forego any claim of comprehensiveness is the speed at which change occurs in networking and on the Internet. When I started this work, in December of 1999, I had intended a chapter on future threats, in which I placed distributed attacks. It was not more than a few months later that several major Web servers were shut down by distributed denial of service attacks. Thus, the future quickly becomes the past.

Another factor is the vast literature on networking and network modeling, which is of immense interest to a statistician and of only marginal interest in network defense. I will briefly touch on this topic in Chapter 2, but it deserves a separate book in its own right.

Since the subject of computer and network security is quite broad, some discussion of scope is in order. First, I will consider what I refer to as "network monitoring." A typical network within a corporation or university is a collection of machines that can communicate with each other and with machines on other networks (the Internet) through a gateway. A network monitor is a system designed to monitor the traffic in and out of the network (or between machines on the network) for the purposes of determining whether the network is working properly and that it is not being attacked from without.

Network monitoring can be as simple as collecting statistics on usage to determine such things as average and peak loads and other measures of the health of the network. It can also include characterizing the kind of traffic on the network as either "normal" (and hence not of concern) or "abnormal" (and hence warranting further investigation). Detecting and characterizing changing activity on the network is of interest, as are sudden deviations from "normal" activity. These ideas will be considered in some detail in Chapters 2 and 4.

An analogy to keep in mind as you read this book is the "envelope" analogy. The information sent across the network is broken into small "chunks," referred to as "packets". Each packet contains addressing information and data. Consider a standard (paper) letter. It contains an address (to and from) and some information as to how the letter is to be handled (e.g., return to sender if undeliverable) as well as content, which resides inside the letter and is generally inaccessible to the mail handlers. A packet is like a letter. It contains addressing and handling information (the "header") and private information (the "data"), which, unlike a letter, is also freely accessible to anyone who wants to look at it (although it can be encrypted for privacy).

Essentially, network monitoring involves measuring statistics on the individual packets sent across the network. One can keep statistics on the headers (the address information on the letter), or one can look at the content to try to infer the intent of the sender. Looking at content is problematic for several reasons:

- High network speeds require extremely fast processing to analyze content.
- Privacy issues often make it politically (or legally) difficult.
- The difficulty of parsing the content is comparable to that of natural language.
- Encryption can make it difficult or impossible to determine the content.

I take the position that network monitoring should primarily concern the address information (header) of the packets, while any content monitoring should be restricted to the individual hosts. Thus, we consider issues of analyzing content or specific individual actions in the chapter on host monitoring, Chapter 5.

Intrusion detection is more specific than network monitoring in the sense that it focuses not only on the detection of "abnormal" behavior but the determination that the behavior is undesirable and/or harmful. In order to make this determination, an intrusion detection system (IDS) must infer both the intent of the activity and the ultimate results of the activity, should it be successful.

There has been a lot of press about computer intrusions in the last few years. Usually the culprits are identified as "hackers," a term that has come to connote a person bent on illegal entry and malicious damage to a computer system. I will refrain from using this term for several reasons. The term "hacker" originally meant someone who was very good at writing computer programs, possibly to the point of obsession. To be a "hacker" was a badge of honor, for it denoted programmers who were at the top of their field. There are still those who hold to the old definition and prefer the term "cracker" for the person intent on damage. Rather than get involved in this battle, I have chosen to sidestep the issue entirely.

Another reason to avoid the term is that it still retains the connotation of a knowledgeable person, when in reality many so-called "hackers" are simply kids (literally or metaphorically) who come across programs that allow them to break into other people's computers. These programs require little skill, assuming the target computer is not well-defended.

Finally, there is the issue of the insider, a person with legitimate access to the computer who, for revenge or gain, decides to damage or otherwise make unauthorized use of the machine. These people are not necessarily expert users and often do no "hacking" in any usual sense of the word. I will refer to any of the above as an "attacker."

This is not a book on how to secure your computer from attack. I will, however, point out various utilities that can help you in this or that are useful for collecting data relevant to intrusion detection. These utilities are all Unix-based, although most of them are also available for other operating systems. All are also available for free. Although there are many commercial products that perform these and other useful security and monitoring functions, I will not cover any commercial products.

There are a number of very good books describing how to secure a given operating system. One I recommend for Linux is Toxen [2001].

The focus of the utilities discussed in this book is almost entirely on collecting data rather than securing a system. Many of the utilities also help to secure a system, and a few are really designed primarily for this task. There are many utilities that have not been listed, due to space limitations, and the interested reader is encouraged to check the Unix manual pages and the Web addresses in Appendix D.

This avoidance of commercial products extends to those designed specifically for intrusion detection. There are several books that cover these, such as anonymous [1997], Escamilla [1998], Amoroso [1999], Northcutt [1999], and Bace [2000]. Also, products change so quickly that anything said about them will likely be inaccurate in a few months. Finally, in order to do a good job of evaluating commercial systems, I would feel the need to acquire them and test them out. This is not an option. Although we have several systems at NSWC that I could evaluate, I decided it best to leave the evaluation of these systems to others. Industry magazines are good places to find such evaluations.

Throughout the book, I have examples of IP addresses and machine names. These should all be considered imaginary, in no way corresponding to a real machine. This is particularly important in the examples of attacks. In no case does an attack example contain the name or IP address of the real attacker or victim, even in those cases where the data come from a real attack.

This book is organized into three sections, covering network basics, intrusion detection, and viruses. Computer professionals with a knowledge of basic networking and TCP/IP can skip most of the first section, whereas statisticians may find this material helpful.

The section on intrusion detection is split into network and host monitoring. Many of the same techniques are relevant to both of these areas, but each has unique features. I will describe some of the more common attacks and some of the approaches to detecting these and other attacks.

The final section covers viruses, worms, and other types of malicious code. The chapter on viruses describes how these operate and takes a slightly different approach to analysis. Rather than focusing on detection, I consider the problem of modeling virus propagation. This is similar to biological virus epidemiology and will make use of techniques from epidemiology. The chapter on trojan programs discusses some common examples of these and the more general problem of covert channels.

Since Unix may not be familiar to all readers of this book, a list of common commands follows.

- **alias** Rename a command (this is actually a shell command rather than a Unix command, but I will ignore this distinction). For example, I have the following on my computers:

  alias ll "ls -lt"

  which allows me to simply type "ll" when I want a time-ordered long listing of a directory.
- **cd** Change the current directory.
- **chmod** Change the mode (read, write, execute permission) of a file or directory.
- **chown** Change the owner of a file or directory.
- **cp** Copy a file.
- **csh** The C command shell (similar to the MS-DOS prompt).
- **echo** Echo the string to the terminal.
- **grep** Search a file for occurrences of a given string.
- **gzip** A file compression utility.
- **head** List the first few lines of a file.
- **kill** Stop the execution of a process.
- **ls** List a directory (similar to MS-DOS dir command).
- **man** Look up a command in the manual pages. I will refer to a manual page as a "man page," which is the standard terminology among Unix users.
- **mkdir** Create a directory.
- **more** View a file one page at a time.
- **mv** Move a file.
- **perl** A powerful language for scanning and extracting information from text files.

- **rm** Remove a file (similar to MS-DOS del command).
- **rmdir** Remove an empty directory.
- **sh** The Bourne command shell.
- **su** Substitute user. Change to another user (for example, root). Some people think "su" stands for "super user," since typing "su" alone is used to change to the "super user," known as "root" in the Unix world. Assuming you know the user's password, you can use "su" to change to that person's account. In particular, root can change to anyone's account. The syntax is

  su username
  or
  su - username

  The "-" makes the shell a login shell, and hence reads any initialization files that are read at login.
- **tail** List the last few lines in a file.
- **vi** A file editor. There are many text editors available. Vi is the classic Unix "visual editor" that is used by many programmers, particularly those who learned programming in the early days of Unix.

There are many books on Unix that provide information on the preceding commands and more. Rather than provide a list, I will leave it to the interested reader to visit a local bookstore.

Part I

Networking Basics

1

TCP/IP Networking

This chapter is intended to provide an overview of networking and the protocols that are most often used for attacks. This should provide the background needed to understand network data and the various attacks described in the following chapters.

The discussion is at a fairly high level. Readers who wish a more in depth discussion of networking are encouraged to investigate one of the many books on the subject. A good place to start is Stevens [1994].

We will start with a brief overview of networking using an analogy of the postal system, a continuation of the envelope analogy discussed in the Introduction. This will provide an intuitive feel for what happens on networks. A program for collecting network data will be described, followed by a discussion of the network layers and encapsulation. The three basic protocols that make up the bulk of IP traffic are described. Packet fragmentation, routing, and domain name service are covered briefly, followed by a few useful utilities for collecting network data.

1.1 OVERVIEW OF NETWORKING

Let us consider the postal system as a high-level analogy to the process that occurs when data (e.g., files, email) are transmitted across a network. To communicate via the postal system one places a message in an envelope and puts the receiver's name, address, and zip code on the envelope. Usually, although this is not required, a return address is put on the envelope. The envelope is then placed in a mailbox. A mail carrier retrieves the envelope and takes it to a substation, where the zip code is read. This code provides the address of the final substation to which the letter is to be delivered. The letter is passed around through various intermediate substations

D. J. Marchette, *Computer Intrusion Detection and Network Monitoring*

until it arrives at the final substation. The final address is then read, and the letter is delivered to the proper address (if all has gone well). Finally, if there is more than one person at the address, someone looks at the letter to determine to whom the letter is addressed and delivers it to that person. All of this is very analogous, at a high level, to the process that occurs when messages are delivered on a network. The one aspect that is not reflected in the network (yet) is the concept of a stamp.

With the preceding discussion in mind, consider what happens when a user decides to send information across a network. For specificity, let us consider email. The user calls up a mail application, types in the message, including the destination email address, and clicks "send." In this instance, let us assume the email address is:

john.doe@someplace.com.

If this were a letter sent through the post office, the handler would read the zip code, which would indicate the city to which the letter should be sent. The Internet has a similar code, called the IP address. This is actually slightly more specific than the standard 5-digit zip code. It corresponds to the complete address, as do the newer 9-digit zip codes. The machine name (someplace.com) gets converted to an IP address, a 4-byte address usually written as four 8-bit numbers separated by periods (these four numbers are referred to as "octets"); for example, 10.10.125.17. The letter is forwarded to the city (or post office within the city), at which point the rest of the address is read. This and further substations are analogous to the routers on the Internet. See Figure 1.1 for a depiction of this.

One advantage networks have over the post office is that they do not require one to remember the numerical zip code (IP address). Instead, a name is provided (someplace.com). To convert the name (which a human can easily remember) to the IP address (which is more convenient for the machine), the network software queries a domain name server (DNS, see Section 1.8), a machine that knows (or knows how to find out) the mapping from name to IP address.

In the simplest case, the email then gets bundled into a packet (analogy: envelope) with the destination IP address included and sent to a router. The application does not actually do the bundling, however. The networking software is implemented in a layered fashion, so that each layer knows just enough to perform its function. This way, the applications need not concern themselves with the details of the networking communications.

We will discuss the network layers in more detail later, but a brief introduction will give a feel for how they work. The application passes the email message to the protocol layer. This layer takes care of such things as making sure that packets are actually delivered and do not get lost. It makes a packet out of the email that tells what protocol is being used for the transmission (more on this later). The next layer, the IP,layer, makes the packet into an IP packet for transmission out across the Internet. The IP protocol is the fundamental "language" of the Internet. Finally, the hardware layer takes care of actually putting the packet out through the hardware and onto the network.

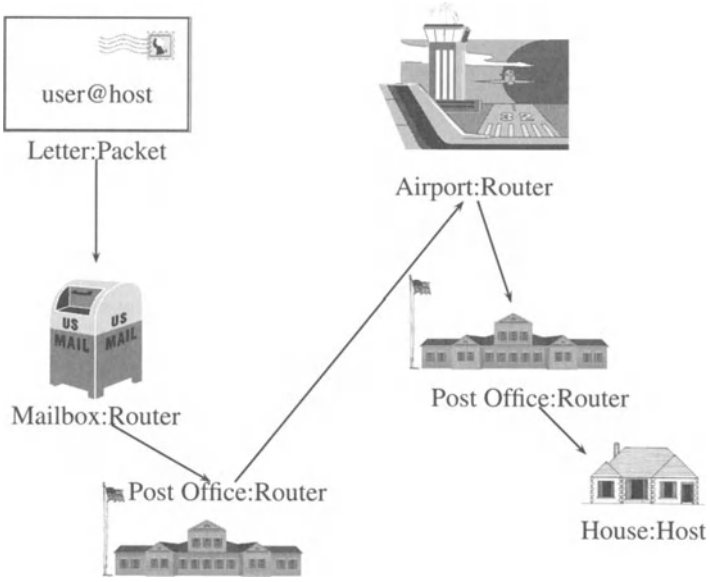

Fig. 1.1 Post Office analogy picture, illustrating the correspondence between network communications and the postal system.

This is a simplification of the process which ignores several layers. The reader is encouraged to investigate Stevens [1994] for more details. It is, however, sufficient for our purposes.

The packet is now on its way to the first router (analogy: the local post office). The router knows the next router to send it to (analogy: the main post office for the destination city). Each router that the packet goes through knows the next router to send to, so eventually it arrives at a router that knows the direct route to the destination machine: someplace.com (AKA 10.10.125.17).

Note that the "routers" of the postal analogy are generally fixed. Letters between two cities go through the same substations, airports, and so on, every time. However, if an airport is closed (for example, by fog), the letter may be rerouted to a train or truck (after all, the mail must get through). This also happens in networks. In fact, it happens much more often in networks, which were designed to be fault-tolerant to an extreme degree. We will see this when we discuss routing.

Now the packet goes back up the network layers. The hardware layer pulls the packet off the network. It passes it up to the IP,layer, which passes it to the protocol layer and finally to the application layer, where the email program (analogy: the local mail carrier) finally reads the "john.doe" of the email address and puts it in the appropriate mailbox.

This overview is a reasonably good approximation of what happens in real life as long as one does not focus too closely on the details. We will now consider

the process in a little more detail. As you will see, the post office analogy breaks down when we start looking at the details.

1.2 TCPDUMP

In order to analyze network data one must be able to collect the data. To this end, various programs, called "sniffers" have been written to capture copies of the packets directly from the network interface. One of the most popular such programs is tcpdump.

The tcpdump program can be set either to collect the data and store it on disk in binary format or provide human-readable output. A typical such output is:

11:00:03.797988 10.10.171.206.1102 > 42.197.95.138.80: S 685673:685673(0) win 8192 <eol> (DF)

This is a TCP packet. We will learn more about these fields in Section 1.5.4, but for completeness I will describe each one in turn. Note that in all the examples the home network will be 10.10.x.x, and all other IP addresses have been "scrambled," or obfuscated, to hide their identity. This obfuscation serves two purposes. First, it acts like the "555-" phone number in movies and television, which protects people from annoying phone calls from confused viewers or pranksters. Second, some of the examples are taken from real attacks, and it is not the purpose of this book to accuse any individual or organization. As will be seen, the "attacker" address is easily manipulated, making it difficult to assign blame for the attack. Also, IP addresses can be reused when the machines that had those addresses are discarded or the company goes out of business. Therefore, if by chance any of the addresses in this book are ever owned by a company or individual, it is safe to assume this is an accident.

The first set of colon-separated numbers is the time the packet was collected. Note that the date does not appear in the timestamp. The usual usage of tcpdump is to encode the date in the filename if the data are stored on disk or to otherwise retain this information for future reference. As we will see later, the date is accessible from binary tcpdump data but is not printed out in the standard human-readable format.

The next two sets of numbers, separated by a ">", are the source and destination IP addresses, with the source and destination port numbers appended on the end. The port numbers are used to set up connections between specific applications and will be discussed in more detail in Section 1.5.4. In this case, the destination port (80) is the port for Web access (http).

The next two colon-separated numbers are the sequence numbers, which provide the packets with a unique ordering. Although the packets are sent out in the correct order, there is no guarantee that they will arrive in order, so some mechanism must be in place to allow the destination machine to properly order the incoming packets. Sequence numbers perform this function.

The "S" in the packet indicates that the "SYN" flag is set (that is that the source machine is requesting a new session to be set up). This is a feature specific to TCP

and will be discussed at some length in Section 1.5.4. A packet with only the SYN flag set is called a "SYN packet".

The "win 8192" indicates a window size of 8192 bytes. Anything in the "< >" at the end represents TCP,options. Finally, tcpdump indicates that the "don't fragment" flag (DF) is set. All of these will be discussed in more detail in the appropriate sections.

The tcpdump program has another useful feature, the ability to filter packets. For example, one can choose to collect only packets that use the TCP protocol. More specific filters are also possible. For example, one can collect all TCP packets having only the SYN flag set from a specific machine to a specific port on another machine (these terms will be defined in the sections to follow). This capability is useful for monitoring for specific known attacks and will be discussed in detail in Section 4.2.

tcpdump can be called with a number of flags to control its operation and output. I will not go through all of them but rather touch on some of the more important ones:

-r file Read from a file of data in tcpdump's binary format. To read from standard input use "-r -". Without the "-r" flag, tcpdump will read from the network interface. Only root has read permission on this interface, so this only works if the user has root permission.

-p Do not put the interface in promiscuous mode. This is useful if you do not want to see any data except that destined for your own machine. Also, it is possible to detect machines with network cards in promiscuous mode, and some organizations (for example, some Internet Service Providers) view such actions as contrary to their security policy. Note: some versions of tcpdump seem to have this option reversed - the default being not to put the interface in promiscuous mode - requiring the "-p" to put it in promiscuous mode.

-w file Write a file in tcpdump's binary format.

-F file Use the filter defined in the file. This will be covered in more detail in Section 4.2.

-s slen This defines the number of bytes (slen) to retain from each packet (the default is 68, which is adequate for the protocols that interest us: IP, TCP, UDP, and ICMP). Larger values of slen allow the collection of packet data, or even the entire packet, if desired. This can be useful for detecting some attacks, but has privacy and security implications.

-tt Display the time as an unformatted timestamp corresponding to the number of seconds since the beginning of time (which in the Unix world is defined to be January 1, 1970).

-n Do not do address conversion. If this flag is not given, tcpdump will attempt to convert IP addresses to names and will also convert some port numbers to application names. This can dramatically slow the execution of tcpdump, as

it requires a DNS lookup (Section 1.8) for each IP address. The result can be lost packets, so use this flag when using tcpdump as a network monitor.

-dd Dump the packet-matching code as a C program fragment. This can be useful for debugging. Similarly, a single "d" dumps the code in a human-readable form, and "ddd" dumps the code as decimal numbers.

For short filters, one can place the filter at the end of the command line, so, for example, to collect only ip packets, use the command:

tcpdump ip

A more involved example might look like:

tcpdump -n "tcp and dst host 10.10.17.25 and not src net 10.10"

In this case, we used quotes to delimit the filter and have specified that we want all TCP packets destined to a given host that does not come from our network (10.10.x.x). Filters will be discussed in more detail in Section 4.2.

It should be noted that although anyone can use tcpdump to view data in a file (assuming they have read permission on the file), only root can use tcpdump to collect live data.

A note on the ethics and legality of sniffers is in order here. One should never install a sniffer on a machine without the permission of the owner of the machine and the security officer in charge of the network. There are serious issues of privacy involved as well as legal issues. A sniffer can provide a copy of every character sent over the network. This allows the reading of passwords (if they are sent unencrypted, which is often the case), email messages, and other private information. Reading these may be considered the same as a wiretap, and hence illegal, in certain circumstances. Although tcpdump can be configured (via the -s flag discussed earlier) to collect a minimum amount of the actual data sent, some data are inevitably collected.

Even if no data are collected, the sniffer provides information such as which Web sites are visited. This information may be considered private in some environments. In some situations, such as work environments, the owner of the network specifies a monitoring policy and provides a security and usage policy detailing the kinds of activity that will be allowed on the network. This policy may allow certain kinds of monitoring, and one may be allowed to install a sniffer for security or research purposes. Check with your network security officer before installing a network monitor on your computer.

It should be noted that a sniffer may not see all the traffic on a network. Traffic between hosts on the network may not travel past the machine hosting the sniffer. This can happen if the sniffer is on a switched network, which is one in which the router acts essentially as a direct connection between any two machines, but it can also happen purely as a result of the network technology. If the destination of a packet is between the source and the sniffer, the sniffer will never see the packet. The destination machine will take it before it reaches the sniffer.

Sniffers can be utilized to enforce computer usage policy as well as for detecting attacks. At NSWC the information security officers decided to monitor traffic looking for inappropriate use of government equipment (porn sites, stock trading, and so on). They used a sniffer to look for certain strings in the content of the packets. However, for whatever reason, it was discontinued. This caused me some relief when I went looking for a book on graph theory. The book (Haynes et al. [1998]) is a compendium of research in an area called "domination", and naturally, one of the words used to describe a particular kind of "domination" in graph theory, is "bondage." Both words appeared on the Web site providing me with the information on the book I was looking for (interestingly enough, my search did not pick up any books that actually were inappropriate). To my amusement, one of the chapters is entitled "Global Domination," which, if it is not one of the key phrases searched for in any inappropriate usage system, should be. I assume, since nobody called me about it, that the filters are indeed turned off (or else I am now on the "enhanced scrutiny" list).

This points out one of the problems with monitoring content. It is difficult to avoid false alarms caused by the many synonyms, idioms, and analogies that are used all the time in English (or any other language). It is possible sometimes to restrict the strings searched for to fairly unambiguous ones (such as "/etc/passwd"), but more general "inappropriate usage" monitors tend to become plagued by false alarms.

This example is relevant to network monitoring from another perspective. In effect, a dominating set for a graph is a set of nodes that are neighbors of all the nodes in the graph. A minimum dominating set is one that contains the smallest number of nodes. It is easy to see applications of this to network monitoring (placement of sensors) and network design (Das and Bharghavan [1997]).

1.3 NETWORK LAYERING

As discussed earlier, TCP/IP networking is implemented in a layered fashion. Many networking books use seven layers to describe the processing. This level of detail is not necessary for our purposes, so the number of levels has been collapsed to four, which is in agreement with the discussion in Stevens [1994]. Each layer will be described in turn, starting from the lowest level and working up to the level seen by the average user.

1.3.1 The Hardware Layer

The hardware (or "link") layer has the task of interfacing with the network hardware. In our postal analogy, the hardware layer plays the part of the mail carrier. This is where packets are physically placed on the network or retrieved from the network. The hardware layer must know details about the specific network interface in the machine as well as what kind of network (e.g., Ethernet, token ring) is to be accessed.

This is where special protocols such as PPP (Point-to-Point Protocol) are implemented for transmission across slow lines such as modems. For the most part, this layer does not play much part in intrusion detection, so we will not cover it in any detail. This is not to say that there are no exploits that utilize specific knowledge about the underlying network protocol, but these are beyond the scope of this work. For a discussion of two such attacks, see Toxen [2001], pages 231–232. These attacks are against the MAC (Media Access Control) address, which is used to route packets to the specific machine on a local area network.

1.3.2 The IP Layer

The IP protocol is the lingua franca of the Internet. It is the underlying "language" that all machines on the Internet must understand in order to communicate. All the higher-level protocols, such as UDP and TCP, are built on this foundation.

The IP layer can be thought of as the letter handler. This is where source and destination addresses are set or read. It also makes sure that the packets have not been damaged in transit. If packets are too large to go across the network in one piece, the IP layer is where they are broken up (see Section 1.6) and subsequently reassembled.

IP is an *unreliable* protocol. This means that it does not attempt to guarantee that packets are delivered. It is up to higher-level protocols (in particular, TCP) to implement any desired reliability.

1.3.3 The Protocol Layer

The protocol (sometimes called transport) layer is where reliability of delivery is implemented. As mentioned previously, the IP protocol does not guarantee that all packets sent will be received, and has no mechanism for handling packets that are lost. It is up to the protocol layer to implement any kind of guarantee.

1.3.4 The Application Layer

The application layer is where the user programs interact with the network. This is where programs such as telnet, FTP, http (Web browsing and serving) operate. Each application can define its own protocol, for example, FTP implements the "file transfer protocol" (hence the name "FTP"), and it is up to the application to manage its protocol. The application layer should be thought of as providing the interface between the user and the network services.

1.4 DATA ENCAPSULATION

One of the important concepts of TCP/IP is the idea of data encapsulation. The application takes the data and prepends an application-specific header, which is used to inform the receiving application of any pertinent information about the data. The application layer then sends the resulting packet to the protocol layer, which

prepends the appropriate protocol header. This header contains any information necessary for the functioning of the particular protocol used. The IP layer then prepends an IP header. The IP header can be thought of as the envelope in our postal analogy. This contains the source and destination addresses and other information required to properly route the packet to its destination. Finally, the hardware layer adds a header (and possibly a trailer) required by the specific network over which the packet is to traverse upon exiting the source machine. Encapsulation is depicted in Figure 1.2.

Encapsulation means that each layer need only know about its own header, which it either adds on or strips off, depending on whether it is sending or receiving a packet. This means that the layers need not be cognizant of the specifics of the other layers, and changes to any one of the layers need not affect the others.

From the perspective of intrusion detection, there are two aspects of the network packets that are important. The headers give information about the source and destination, what protocol is used, options about how the packet should be routed, and what service or user program is the ultimate destination of the packet. The data field contains the actual data transmitted and so contains things such as email addresses, files being transferred, and passwords (for example, in a login session). Although many intrusion detection systems look for strings within the data, there are some systems that look exclusively at the header information (SHADOW, discussed in Section 4.4, is one example of such a system). Also, many of the interesting statistical questions deal with header information. We will look at these in some detail.

1.5 HEADER INFORMATION

We now consider the different headers in detail. We will not be concerned with the hardware-specific headers. Although these may be of interest for certain kinds of network reliability and other analysis tasks, and can be of interest in detecting certain kinds of sophisticated attacks, we will leave discussion of these to more specialized texts.

For all the headers in the following figures, each row corresponds to 4 bytes of information. The fields of the headers are (unless otherwise specified in the text) either 32-,16-,8- or 4-bits, as indicated by the size of the box. For example, Figure 1.3 depicts the IP header. The first row consists of two 4-bit fields, one 8-bit field and one 16-bit field.

1.5.1 IP Packets

IP stands for Internet Protocol and is the fundamental protocol of the Internet. Essentially all packets sent over the Internet are IP packets.

As seen in the encapsulation figure (Figure 1.2), the IP layer is the one constant in the layers of TCP/IP. The hardware layer is specific to the network hardware and the specific local area network to which the machine is connected. As we will see, there are several protocols implemented at the protocol layer, and obviously

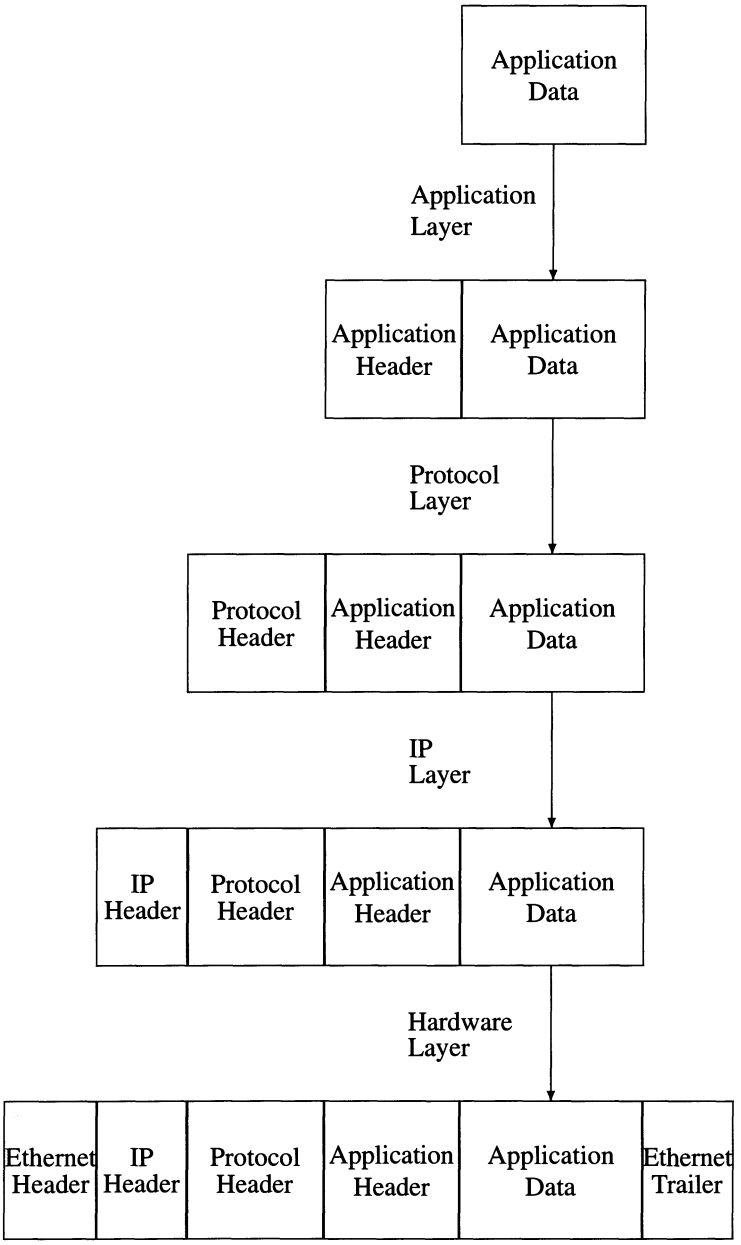

Fig. 1.2 Data encapsulation. Each layer prepends a header onto the packet as it is passed down the IP stack.

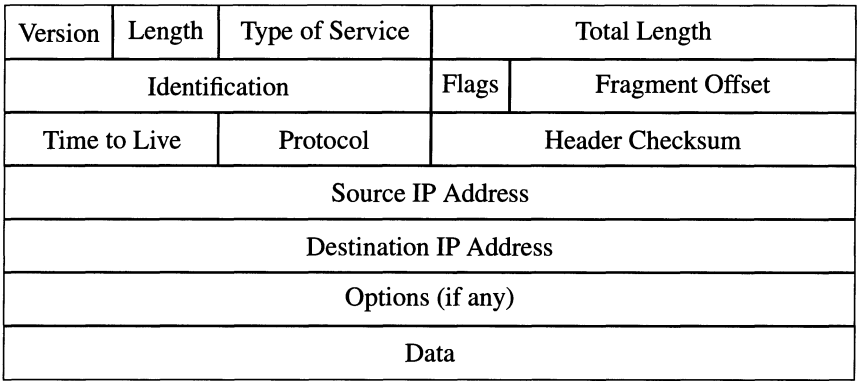

Fig. 1.3 The IP header.

there are many different applications. It is only the IP layer that is constant. All TCP/IP packets have an IP header.

Before considering the header fields in detail, we must address byte order. Consider an integer that is two bytes long. For some machines (so-called "little endian") the high-order bits are in the leftmost byte, the low-order bits on the right. For all network headers, the "big endian" convention is used. High-order bits are on the right. So a 4-byte number has bits 0–7 first, then bits 8–15, ..., 24–31. This convention is consistent regardless of the convention of the machine constructing or reading the header. This byte ordering is called the network byte order.

The IP header is depicted in Figure 1.3. The first field, the 4-bit version, is always set to 4 (for the current version, IPv4). The next version will be IPv6, which is not yet in wide use as this book is being written. The header length field contains the number of 4-byte words in the header. Since this is also a 4 bit field, this means that there can be no more than 60 bytes, or 15 4–byte words in an IP header. If no options are set, the value of this field will be 5.

The type of service field (8 bits long) is used to indicate a preference for how the packet should be routed. The first three bits are ignored. Only one of the next four bits should be set. The bit that is set indicates that the packet should be routed according to one of the following criteria.

- Minimize delay.
- Maximize throughput.
- Maximize reliability.
- Minimize monetary cost.

The final bit must be set to 0.

The total length field contains the total length (in bytes) of the IP datagram. Thus, no packet can be longer than 65,536 bytes.

Each packet has an identifier contained in the identification field. Packets from a given machine that are likely to be on the network at the same time should have distinct identifiers. Obviously, these are ultimately reused.

The flags (3 bits) and fragment offset (13 bits) fields will be discussed in more detail in Section 1.6. Suffice it to say that they are used when a packet is too large to traverse a given network and must be broken into smaller packets.

The time-to-live (TTL) field is used to keep packets from being immortal. Each router that handles a packet decrements the TTL field. If this field reaches 0, the packet is dropped (ceases to exist) and a message is sent back to the source computer (see Section 1.5.2). In this manner, no packet can survive through more than 255 routers, and most packets have an initial TTL much smaller than this. As we will see in Section 1.9.5, this functionality can be used to determine the route a packet can take and to map the network.

The protocol field tells which protocol is used by the protocol layer. This is the one place where the IP layer looks at the header passed down to it by the protocol layer. Different protocols have different header lengths and fields, and the protocol field is used to indicate which header is encapsulated in the packet.

The header checksum is used for error detection. It is computed over the IP header only. The checksum is calculated by treating the entire header as a collection of 16-bit numbers. The checksum field is first initialized to zero. The one's complement sum is taken of the header. The one's complement of this value is stored in the checksum field. Upon receipt of an IP packet, the one's complement sum of the header is taken (again as a series of 16-bit numbers), and, since the checksum is the one's complement of the sum of the rest, this number should consist of all ones. If it does not, the header has been corrupted and the packet is dropped. The IP layer does not generate an error message in this case, but merely discards the packet.

The source and destination IP addresses are 4-byte numbers. These are also in big-endian format, and so are stored low byte first: the IP address 10.11.127.13 is stored as 13 127 11 10 in the 4-byte field.

The (optional) options field allows the selection of a number of possible routing and/or recording choices. The possible options are:

- **Record Route**. If this option is set, the IP address of each router the packet goes through is added to the end of the IP header, recording the route taken by the packet. Unfortunately, since the IP header has a limited capacity, only a maximum of nine IP addresses can be stored.

- **Timestamp**. This is similar to the record route option except that it records the time each router receives the packet. It can be set to record only the times, or the times and IP addresses, of each router. A list of up to four IP addresses can be provided, in which case only those routers matching the list will record arrival times.

- **Loose Source Routing**. This specifies a list of IP addresses through which the packet must be routed. It does not restrict the packet from traveling through other routers in addition to those on the list.

- **Strict Source Routing**. Like loose source routing, this specifies a list of IP addresses through which the packet must be routed. However, only the addresses on the list can be traversed; no other routers may be used.

Not all the options are implemented by all machines and routers, and for the most part they are not used on modern networks. Some of them can be security threats. For example, source routing can be used to implement a covert channel. Suppose John wishes to send proprietary data to Maria without it being obvious that the data are leaving his company. If Maria owns a router, John can simply route his packets through Maria's router using source routing, thus allowing Maria to see whatever John sends. Thus, when John sends email to his boss discussing the bid their company will be making on a contract, Maria sees the information as it passes through her router. Although this kind of activity is easily detected by a security analyst, it is generally undetectable by John's boss, or other coworkers, and hence may go unnoticed in many organizations. For this reason, among others, source routing is often disabled on modern networks.

We now turn to the three most common protocols on the Internet.

1.5.2 ICMP Packets

The Internet Control Message Protocol (ICMP) is, as its name implies, a protocol for sending messages related to the control of the Internet. It is used to send error messages or other information pertinent to the functioning of the network. Figure 1.4 depicts the ICMP header.

The type and code fields are used to identify the type of message sent. Table 1.1 contains a description of the types currently implemented. See Stevens [1994] for more information on the types and codes currently implemented. The data field of ICMP packets can contain extra header fields for specific types and codes. For example, Figure 1.5 shows the header used for ICMP echo requests and replies. This will be used when we look at the Loki trojan in Section 7.4, which uses ICMP packets to implement a hidden login session.

ICMP is the protocol in which the ping program is generally implemented. Ping is a program that is used to determine if a machine is alive on the network (see Section 1.9.1). A series of *echo requests* are sent to the computer, and the program looks for *echo replies* returned by the computer. It keeps track of how many packets elicited responses (giving a measure of packet loss on the network)

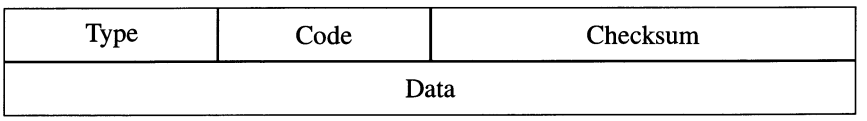

Fig. 1.4 The ICMP header.

Type	Code	Checksum
identifier		sequence number
Data		

Fig. 1.5 The ICMP header for echo requests and replies.

Table 1.1 ICMP message types.

type	Description	Purpose
0	Echo Reply	Query
3	Destination Unreachable	Error
4	Source Quench	Error
5	redirect	Error
8	Echo Request	Query
9	Router Advertisement	Query
10	Router Solicitation	Query
11	Time Exceeded	Error
12	Parameter Problem	Error
13	Timestamp Request	Query
14	Timestamp Reply	Query
15	Information Request	Query
16	Information Reply	Query
17	Address Mask Request	Query
18	Address Mask Reply	Query

and how long between each request and subsequent reply, providing a measure of the distance to the machine (or load on the network).

1.5.3 UDP Packets

The User Datagram Protocol (UDP) provides a mode of communication between applications. A single datagram is produced for each output of an application. There is no guarantee that a packet will reach its destination, and there is no built-in mechanism to detect lost packets. This means that the protocol is not reliable, in the sense that TCP (to be discussed in Section 1.5.4) is.

The datagram consists of the UDP header (Figure 1.6) and data generated by the application.

UDP implements the concept of "ports" used to communicate with different processes. The port numbers are identifiers used to mark the different processes. The ports are a logical construct rather than a physical one. Each application process selects one or more ports through which it will send information and one or more at which it will listen for incoming information. Since the port numbers are 16-bit numbers, there are a maximum of 65,536 ports.

The source port and destination port indicate which application is sending and receiving the packet. The length field is the total length of the UDP datagram in bytes. This field must have a value of at least 8, since that is the length of the header. The checksum is calculated in the same manner as in the IP packet, except that it is calculated for the entire datagram, including the data. This provides a measure of error checking to determine whether the packet was corrupted in transit. If a packet is determined to have been corrupted (fails the checksum test) it is dropped. This means that the packet is ignored, not sent up to the application layer, and no error message is generated.

The checksum is optional, unlike the IP checksum, but should always be used. As with IP, the packet is silently discarded if the checksum indicates that the packet has been modified. If checksums are disabled, no test is made, and all packets are sent up to the application layer.

Since the UDP checksum is computed over both the data and the header, and like the IP checksum uses a 16-bit word, it must be able to handle data of an odd length. It does this by padding with a zero if necessary.

The UDP checksum is different from the IP checksum in another respect. It prepends a "pseudo-header" consisting of the source and destination IP addresses, the 8-bit protocol from the IP header, and the UDP data length to the UDP header prior to calculation of the checksum. This adds another layer of assurance that the packet was properly delivered and unmodified.

1.5.4 TCP Packets

The Transmission Control Protocol (TCP) is the protocol that implements reliable communication on the Internet. Rather than simply sending packets from one machine to another, as in UDP, TCP implements the concept of a *connection*. A connection can be thought of as a communication channel, where both sides have

Source Port	Destination Port
Length	UDP Checksum
Data	

Fig. 1.6 The UDP header.

agreed on the communication, and mechanisms are put in place to ensure that all packets arrive unchanged at their destination.

Reliability is provided by several key features unique to TCP. First, TCP acknowledges the receipt of each packet. It maintains a timer, and if the acknowledgment is not received within the predefined time limit, it resends the packet. Second, since IP packets can be received in any order, it includes a unique number for each packet, ensuring the receiving application can reconstruct the correct order and also detect when packets have been lost and hence will eventually be resent. Finally, since each process has a finite buffer space in which to store packets, TCP ensures that the sending machine never sends too much data to be stored in the receiver's buffer.

The TCP header is depicted in Figure 1.7. Like UDP, it has source and destination ports, which indicate which application is the ultimate recipient of the packet.

The 32-bit sequence and acknowledgment numbers are used to ensure that the packet ordering is maintained and that no packets are lost. When a connection is first initiated between two machines, the initiating machine provides an initial sequence number, which is subsequently incremented throughout the session, providing an ordering to the packets. We will discuss this in more detail later.

The length field, like the IP length field, is the length of the header in 32-bit words. It is a 4-bit number, hence restricting the header length to at most 60 bytes.

The reserved field is a 6-bit area reserved for future extensions to TCP. Since the advent of IPv6, it is unlikely that this will ever be used.

The flags are bit values within a 6-bit field, used to implement and control the connection. Their values are, in the order they appear in the bit field:

- **URG** indicates that the urgent pointer is valid (see below).

<table>
<tr><td colspan="3">Source Port</td><td>Destination Port</td></tr>
<tr><td colspan="4">Sequence Number</td></tr>
<tr><td colspan="4">Acknowledgment Number</td></tr>
<tr><td>Length</td><td>Reserved</td><td>Flags</td><td>Window Size</td></tr>
<tr><td colspan="3">Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="4">Options (if any)</td></tr>
<tr><td colspan="4">Data</td></tr>
</table>

Fig. 1.7 The TCP header.

- **ACK** the acknowledgment number is valid. This is used to acknowledge receipt of a packet.
- **PSH** this indicates that the data should be "pushed" up to the application as soon as possible.
- **RST** reset the connection. This indicates that something has gone wrong, and the connection should be broken off.
- **SYN** synchronize the connection. The sequence numbers are synchronized so that each end knows the order of the subsequent packets. The SYN flag is used to initiate a connection.
- **FIN** finish the connection. This is used to indicate that the sender is finished sending data and that thus the connection (in this direction) should be closed down.

We will discuss the flags in more detail later.

Window size is the number of bytes that the receiver is willing to accept. This is the size of the transmit or receive buffer. The window size can be used to increase throughput for file transfers and other applications.

TCP, like the other protocols, includes a checksum in the header. In the case of TCP, the checksum is mandatory. It also utilizes a pseudo-header in the same manner as UDP, described in Section 1.5.3.

The urgent pointer is a way for an application to send emergency data to the receiver. For example, when a user aborts a program (by hitting Control-C), the application can notify the receiver that the next few bytes of data are important and should be handled as such. This is implemented by setting the urgent flag and placing in the urgent pointer the offset to be added to the current sequence number to indicate the last byte of urgent data.

There are a number of other options available in TCP. A complete list is beyond the scope of this discussion. The reader is encouraged to check Stevens [1994] or other books on TCP/IP for details about the possible options.

1.5.4.1 TCP connections A TCP connection is first initiated by a machine sending a packet with only the SYN flag set. This is analogous to the machine asking "hello, are you there?". The receiving machine then sends a packet with both the SYN and ACK flags set, acknowledging the initial SYN, analogous to the reply: "yes, I'm here, let's talk." Finally, the initiating machine sends a packet with only the ACK flag set, indicating that the connection is now in place.

Let us consider this "three-way handshake" in more detail. The initial SYN packet must have no flags other than the SYN flag set, and it must contain a sequence number. This is the number to be used from now on as the initial number for the sequencing of packets. Each subsequent packet sent from this machine will have a sequence number incremented from the previous one. The receiving machine then replies with only the SYN and ACK flags sent and with an acknowledgment number that is the original sequence number incremented by one. This is the next sequence number it expects to see from the first machine. It also adds a sequence number of its own. The first machine, when it acknowledges this

packet, sends this second sequence number, incremented by one, thus indicating the next sequence number it expects to see from the second machine. This is illustrated in Figure 1.8.

Once the three-way handshake has been completed the TCP connection is open for communication both ways. The sequence and acknowledgment number keep track of the order of the packets and allow the detection of packets that are lost along the way.

The connection is closed via a four-way handshake (or pair of two-way handshakes, if you prefer) of FIN/ACK packets, closing the two directions of the communication channel. One host sends a FIN packet, which is acknowledged by the other host via a FIN/ACK. This closes communication from the first host to the second. The second can continue sending packets to the first, however, until it sends its closing FIN packet, which is acknowledged by a FIN/ACK.

A typical TCP session might look like the one depicted in Table 1.2. It begins with the three-way handshake. There are a series of pushes and acknowledgments, and then the two closing FIN handshakes. Note that the machines do not have to acknowledge every PSH. Instead, an acknowledgment indicates receipt of all packets up to the one acknowledged. Also, note that after one side closes the connection, the other side can continue sending data until it decides to close its connection.

If TCP is so much more reliable than UDP, why does UDP exist at all? Why not use TCP exclusively? The main reason is the overhead involved in ensuring the reliability. For applications where reliability is not that critical, UDP can be faster, and require fewer packets, than TCP. For example, I was involved with a project to automatically find objects in a video (for example, tanks in the desert). This was implemented on a cluster of nine Linux machines, where the processing was

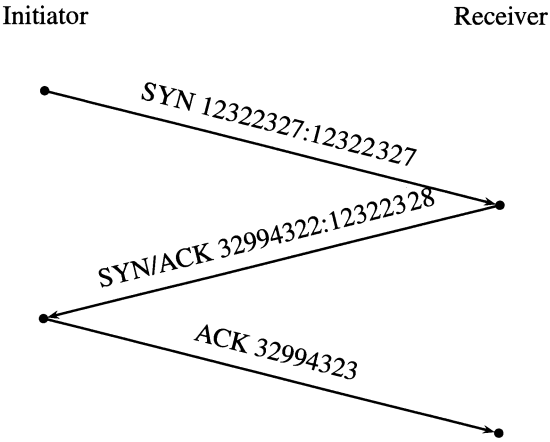

Fig. 1.8 The TCP three-way handshake. The initiator starts the connection with an initial SYN flag. The receiver acknowledges the SYN, and the initiator acknowledges the acknowledgment.

distributed across the machines using UDP packets. Any corrupted packets would merely cause a slight error on a piece of one frame, and the speed requirement was such that this was considered a small price to pay (in fact, in several months of processing, we have never noticed a problem). The system runs at 15 frames a second (1/2 of real time) on eight 450 MHz Pentium III microprocessors (the ninth computer simply manages the processing and displays the results to the user).

1.6 FRAGMENTATION

If we send a large letter through the postal system, we simply pay more for stamps. On a network, the letter (packet) gets broken up into smaller packets, which get sent along and then reassembled at their destination. This process is called fragmentation.

Fragmentation is controlled by the flags and fragment offset in the IP header 1.5.1. There are two flags that control the fragmentation, denoted DF and MF. If the "Don't Fragment" (DF) flag is set, the packet will not be fragmented. This means that if it arrives at a router that wants to fragment the packet, the packet is

Table 1.2 A "typical" TCP session.

Host 1	Host 2
SYN	
	SYN/ACK
ACK	
PSH	
PSH	
PSH	
	ACK
	PSH
	PSH
ACK	
PSH	
	ACK
FIN	
	FIN/ACK
	PSH
	PSH
ACK	
	FIN
FIN/ACK	

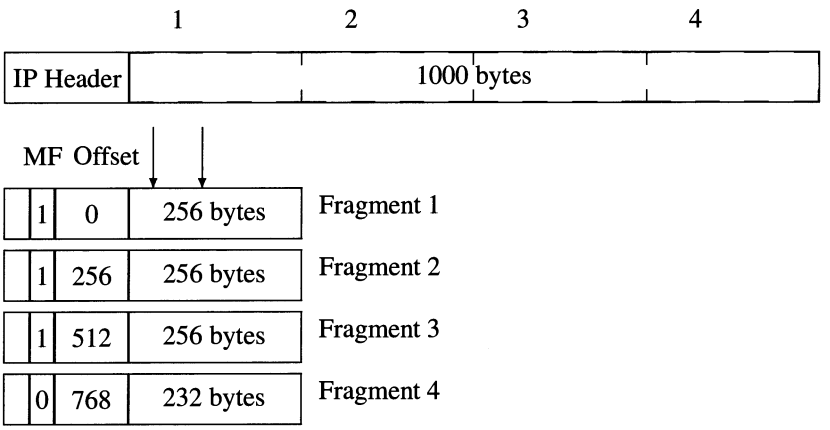

Fig. 1.9 An illustration of packet fragmentation. A packet arrives at a router which has a maximum packet size of 256 bytes. The packet is broken into packets of this size, each (except the last) having the "more fragments" (MF) bit set and the offset set appropriately.

not forwarded and an ICMP error message (type 3, code 4: "fragmentation needed but don't fragment bit set") is generated and sent back to the originating machine.

If the DF flag is not set, the packet is broken up into smaller packets, each small enough to be forwarded. Each packet, except the last, has the "more fragments coming" (MF) bit set.

The fragment offset field is used to indicate where each fragment belongs in the reconstructed packet. The first packet has this field set to zero. Subsequent packets have the field set to the number of bytes that come before the fragment. This is illustrated in Figure 1.9.

Upon receipt by the destination machine, the fragments are reassembled into the original packet. The placement of the fragments in the reassembled packet is governed by the fragment offset since the fragments are not guaranteed to arrive in order.

1.7 ROUTING

The Internet is a loose collection of machines with no global authority ensuring that packets are delivered or even that machines know where to send packets. Providing a direct route between all machines on the network might be practical for a network of a few tens of machines, but for the Internet this is simply not possible. Instead, machines must be able to determine for themselves the best route to use to send packets to a particular destination.

To this end, each host maintains a routing table, which is basically a list of destinations (hosts or networks) and gateways (routers) to use as the first hop to the destination. This list is fairly stable, changing only occasionally, compared to the number of times it is accessed. It provides the host with an address (in our postal analogy, a mailbox or local post office) to send packets destined for a given machine.

We will consider this process at only a very high level. The basic ideas presented here can be found in Stevens [1994]. Networking books can provide a more detailed description for those interested in routing.

Suppose host S wishes to send a packet to host D. First, S checks its routing table to see whether host D (or its network) is on the list. If it is, it obtains the gateway address and sends the packet to this address. If there is no match, it checks to see whether there is a default entry in the table. This has a router associated with it to which all packets should be sent, if no more direct route can be found in the table. If you configured your machine yourself when it was first placed on the network, you were asked for a "default gateway". This is the router to which most of your outgoing packets will be sent. If there is no default, then a "network unreachable" or "host unreachable" message is sent (if S is a router forwarding on a packet) or the application is notified that the packet cannot be sent.

A digression is appropriate here. Up to now we have considered only the information in the IP header. Suppose, in our example, that S wants to send a packet to D and the routing table indicates that the packet should therefore be sent to router R1. How is this done? How is the address for R1 put into the packet so that the network can deliver it? Obviously, the destination IP address cannot be used since this would overwrite the intended destination address. The solution is to note that the packet is going out over a particular network interface and must be encoded with the network address to which it is to be delivered. This is the purpose of the Ethernet header depicted in Figure 1.2. Thus, the routing table provides a network address, rather than an IP address, to be used to route the packet.

Since the routing table is so important to the functioning of the network, there must be mechanisms in place to initialize the table and to update it as needed. At boot time, a router will broadcast a series of ICMP packets which advertise its availability. These let hosts on its networks know that it is up and ready to receive. When a host boots up, it broadcasts ICMP solicitation packets, which then cause any routers on the network to respond, letting the host initialize its routing table.

The routing table can also be updated by a "redirect" message. Suppose in our example, router R1 notes that the next router, R2, can be reached directly by our host S. It then sends a "redirect" message to S informing it of this fact. S updates its routing table so that all future packets to D can be sent directly to R2, bypassing the unnecessary router R1.

In addition to the preceding mechanism, routers can talk to each other using for example the routing information protocol (RIP). This allows them to inform each other of changes in the networks they connect to so that better routes can be computed and routes dropped if they are no longer available through a particular router.

1.8 DOMAIN NAME SERVICE

In order to send a letter, you need to know the address, and the same is true for a packet. People have a much easier time remembering names, and have a penchant for naming their machines, so it is convenient to have a "human readable" address for each machine, such as "dvader.wallaby.org." Unfortunately, the network needs

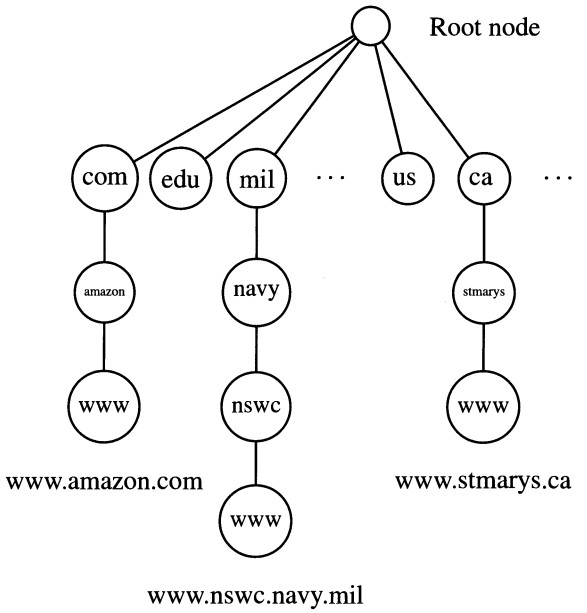

Fig. 1.10 The hierarchical organization of the domain name service. The first level domains are referred to as top-level domains, the next level as second-level domains.

IP addresses, such as 10.10.138.42. The way the mapping is made from the easy-to-remember names to the IP address is through the Domain Name Service (DNS).

DNS is a distributed database. No single machine contains all the information necessary to make the mapping. This is important for two main reasons. It means that there is no single failure point, and it also means that a network the size of the Internet can function. If every machine had to query a single machine (or small number of machines) every time it wanted to send a packet, there would be a huge bottleneck that would make large networks impossible. DNS is the solution to this problem.

Each site (for example, a company, university, or Internet service provider (ISP)) maintains a database of all the machines (hostname and IP address) that are on its network. This information is hosted on a "domain name server" (also abbreviated DNS), a machine that maintains the database for the site and provides the information as needed.

The DNS is organized hierarchically, as depicted in Figure 1.10. Each top level domain is given a 2– or 3–character designator (there is at least one 4–character designator, arpa, which we will not discuss). Table 1.3 lists the 3–character domains (as given in Stevens [1994]).

In addition to the generic domains, there are two-character country codes, such as us (United States), ca (Canada), and so on. See Appendix C for a listing of the country codes as of the time of this writing. Each country may have a convention for other domains (such as for states in the U.S. or educational institutions in the U.K.). The rest of the name can be fairly arbitrary, with some restrictions on legal

characters, up to 63 characters in length (capitalization is ignored). If one were so inclined, (and the administrator in charge of the "someisp.net" DNS allowed), one could have a machine called:

givemeyourtiredyourpooryourhuddledmassesyearningtobefree.someisp.net

It is unlikely that one would be allowed to choose such a name. It is also hard to imagine that one would want to. On the other hand, it is quite common for us at NSWC to see packets that appear to come from machines named things like

will.work.for.food.com

or

dazed.and.confused.org

or somewhat less appropriate phrases, often involving bodily functions. This is probably a result of someone inappropriately manipulating an insufficiently protected DNS.

To illustrate this, I looked through the SHADOW logs (see Section 4.4 for a description of the SHADOW system) for three days and came up with six apparently hacked machine names. The two that did not contain profanities and hence can be reproduced here were:

dont.blame-me.im-a.beginner.org

and

is.not.the.dumbe.st,

the latter using a splitting of the word "dumbest" to spoof originating in Sao Tome.

Table 1.3 Generic domain designators.

Domain	Description
com	commercial organizations
edu	educational organizations
gov	some U.S. government organizations
int	international organizations
mil	U.S. military
net	networks (Internet service providers)
org	non-profit organizations
arpa	old style arpanet
nato	NATO field

There are two common goals in attacking DNS servers. The first is to map IP addresses to funny names, as just indicated. This is (relatively) harmless. The second is to change the IP address for a given domain name. This is much more serious. The attacker changes the table so that when a machine requests the IP address for mybank.com, it gets the IP address for villainsrus.net. The attacker's machine is set up to mimic the bank's and waits for unsuspecting users to start entering account and password information.

This happened to a colleague of mine. He tried to show me his Web page and ended up at the page of some rock band instead. He thought his page had been hacked, but instead it was the DNS server. In this case, the "attack" was mostly harmless, at least from his perspective, but it was quite annoying, particularly when he thought he had lost everything on his Web site. This apparently innocuous prank could have been quite costly if he had been running a business from his Web site instead of simply providing information about his area of expertise.

Although it is true that nobody maintains everything, there must be a place to start a DNS lookup if the information is not maintained locally. There are a small number of computers that know which name servers are responsible for which domains. There are about a dozen of these "root" servers, which each DNS must know about. These servers maintain a list of the responsible name servers for the various top-level domains. As illustrated in Figure 1.10, these maintain lists of secondary name servers and on down the tree.

When you want to send a packet to www.widgetsrus.com, your machine first checks to see if it already knows the IP address (for example, if it is in your local host table or you have recently done a DNS lookup for that host). If not, then it queries the local DNS. This machine keeps a cache of recent queries as well as its local table, so it checks these to see if it knows the IP address or if it knows which DNS to go to for the information (say, from a recent query on ftp.widgetsrus.com). If not, it queries one of the root servers, which tells it where to start on the tree. Eventually (usually after just a few steps), it obtains the information from the appropriate DNS.

How does the attacker go about changing the DNS entry? Obviously, one way would be to gain access to the host that is acting as the domain name server and directly change the lookup table. There is a much easier (and safer) way, however. It is called DNS "cache poisoning" (see Klein [1999]).

To understand how this works, it is important to know that DNS lookups work via UDP. Since UDP is connectionless and stateless, it is easy to spoof UDP packets. The idea is to use a UDP packet that purports to be the answer to a DNS lookup but contains altered information. The target updates its cache and from then on provides the wrong address when queried.

There are several reasons for doing this. Obviously, some people do it just to show they can. Others do it for profit, as described earlier. Another way to profit is to have a site that earns money through advertising based on the number of hits to the site. Redirecting a popular site to yours can artificially increase the number of hits. Redirecting the site of a competitor can increase your sales. Finally, one may wish to sabotage a site by redirecting it, thus causing loss of customers or embarrassment. It goes without saying that all of these are both illegal and wrong.

One can actually steal a domain name by forging an email authorizing a change to the domain name registry. Although one can take steps to avoid this, many people (particularly those new to the Internet) do not. Thus, an attacker can in effect take your domain name away from you.

Finally, one can register a domain name similar to the one owned by someone else. When companies decide to go on the Internet, or when they change their name, they will often buy up all the domain names that are at all relevant to their company (if they can). There was a story that illustrated this (and its futility) when GTE and Bell Atlantic merged to form Verizon. Verizon registered (Goldstein [2000], pp. 16–17) over 700 domain names, some of them legitimate sounding, such as verizonwireless.net, as well as others, such as verizonstinks.net, aimed at stopping critics and disgruntled customers from using the company name. The people at 2600 (a group of self-described hackers) found one (slightly rude) that had been missed and promptly registered it. After some legal scuffling, 2600 registered the domain name

verizonshouldspendmoretimefixingitsnetworkandlessmoneyonlawyers.com

This anecdote illustrates an important lesson: although it may be a good idea to register a few names to protect yourself from copy cats, you cannot think of them all. In fact, trying to can just make things worse.

1.9 MISCELLANEOUS UTILITIES

This section includes a few useful utilities for investigating networks. These include information-gathering tools such as ping, traceroute, and whois, as well as ssh, a program to allow secure logins.

1.9.1 ping

Ping, the Packet Internet Groper, originally written by Mike Muuss, is used to measure the round-trip travel time between two machines. (Actually, the name "ping" comes from an analogy with submarines and sonar, the expansion as an acronym came later).

There are a number of implementations of the ping utility, but I will discuss the most common (ICMP) implementation.

The standard usage is

ping host

Several "echo request" ICMP packets are sent to the host. The host replies with "echo reply" (unless a firewall or other security measure denies this or the machine is not responding or nonexistent), and the time between packets is computed. This gives an estimate for the time it takes for packets to transit between the machines. Packets will be sent until the user kills the program (one can specify the number of

packets to send by using the command line option "-c count"). One can also adjust the time between packets and the size of the packet. As we will see in Section 4.3.1.3, this latter capability is a boon to attackers (although most implementations of ping limit the size to a legal one, defeating this particular approach to mounting an attack).

Network engineers can use ping to analyze problems on a network. It is also useful to users for determining whether a particular machine is down or the network itself is down. On one of the networks that I use, it is not uncommon for connectivity to the outside (for example, for Web surfing) to be down for no apparent reason. In the past, I could use ping to determine whether it was the network that was down or the Web site I was trying to reach (by "pinging" other hosts on the inside and outside). This is no longer possible since many sites no longer allow "echo requests" or "echo replies" in or out of their networks.

Ping has been useful to attackers to determine whether a particular machine is up, to map a network (see Section 4.3.2.1), or to mount an attack (see Sections 4.3.1.3 and 4.3.1.6). This is one of the reasons that many sites do not allow these packets into or out of their network.

A related utility (on some machines) is fping, which allows the user to specify multiple hosts and returns the results in a more machine-readable format. See the man page for more information. Also see

http://ftp.arl.army.mil/~mike/ping.html

for some interesting background (and humor) about the program.

1.9.2 nslookup

As mentioned earlier, the IP address of the destination computer is needed for network connections, but people use easily remembered names for the computers. We have seen in Section 1.8 how computers obtain this information. The "nslookup" command is how a user can map between names and IP addresses.

The simplest usage is

nslookup machine

where "machine" is either a machine name or IP address. nslookup will return something like:

```
Server: resolver.myisp.com
Address: 10.10.1.1

Name:  www.amazon.com
Address: 208.216.181.15
```

(assuming "machine" is "www.amazon.com").

In addition to the preceding usage, nslookup has an interactive mode. This is entered simply by typing

nslookup

This will give you a prompt (probably ">"). Now you can type machine names (or IP addresses) one at a time, and it will resolve each one. There are other commands available; type "help" at the prompt for a listing of the available commands.

1.9.3 whois

The "whois" directories give information on the owner of a particular domain name or IP address. There are a number of implementations of whois, but the easiest ones to use are Web-based servers, such as those found at

```
http://rs.internic.net/whois.html
http://www.nsiregistry.com/whois/
http://www.iana.org/cctld/cctld-whois.htm
http://www.betterwhois.com/
http://www.networksolutions.com/cgi-bin/whois/whois
http://www.whois.net/
```

Also, SHADOW (Section 4.4) contains a utility for doing whois searches.

The whois servers have the look and feel of an Internet search engine. There is a field to type your query (such as "microsoft.com" or "10.10.132." - don't forget the final "."), and the server will return an address for the owner of the domain name or network. One problem is that each server tends to search a subset of domain name space (for example, .com, .org, and .net only), so one may have to do several searches. In particular, most whois servers do not provide information on U.S. military (.mil) domains. The utility in SHADOW allows the selection of the domains to search, making this relatively painless.

One site

http://www.cybergeography.org/,

uses whois lookups to map domain names to physical addresses and then plots these on maps. This is not perfect since, for example, a company may use its corporate address for its domain names, regardless of the actual location of the machines. Thus, the "cybergeography" is really a display of the locations of the owners of domain names rather than a map of the actual physical locations of the machines.

1.9.4 ssh

Logging on to machines outside your network can be hazardous due to the proliferation of sniffers that can obtain your user name and password without your knowledge. One way to avoid this is to encrypt the information. This is the function of the secure shell (ssh) utility. ssh, and its related copy command scp, first negotiates an encrypted session between the machines. This sets up a secure channel through which information, such as the password, can be (relatively) safely

transmitted. Since the whole session is encrypted, any information transmitted is protected from observation.

ssh (actually the encrypted copy utility scp) is used by SHADOW (Section 4.4) to transfer the hourly files from the sensor to the analysis station. This way, even data from remote sites can safely be transferred without fear that someone will capture the data. This is important since the data consist basically of sniffer files, and we do not want others using our sniffer against us.

One of the nice things about ssh is that other TCP connections can be forwarded over the secure channel. Thus, one can run X Windows connections (for example, use Netscape to browse the Web from the remote machine).

Since ssh uses encryption, its availability may be restricted outside of the U.S. (although these laws appear to be changing). It supports several different methods for authentication and encryption. A full description is beyond the scope of this book. The man pages and documentation that come with ssh should be consulted for more information.

The Web address for ssh is

http://www.ssh.fi/

1.9.5 traceroute

Traceroute, as its name implies, is a utility for tracing the route from one host to another. It is similar to ping in that it sends a series of packets to the destination and computes some simple statistics on the returning packets. In fact, a version of traceroute could be implemented using ping. However, UDP packets are used instead, as will be described later. One can force traceroute to use ICMP packets via the -I option (see the following).

Recall that the IP header contains a field called the "time-to-live" (TTL) field. This field is decremented at each router. When a router decrements a TTL field to 0, it sends a "time exceeded" ICMP packet back to the originating host. The idea behind traceroute is that if you know the original value of the TTL field, you then know how many routers the packet passed through before the final router. The key to making this really useful is the fact that the "time exceeded" ICMP packet contains the IP address of the final router as the source address.

Traceroute works by sending packets with increasing TTL values and reporting the IP addresses of the routers. The TTL increases from an initial value of 1 until the destination machine responds, indicating that the full route has been traversed.

How does traceroute know that it has reached the destination? The fact that no router has responded with a "time exceeded" packet might be the result of lost packets, rather than the packets reaching the destination. Traceroute solves this by sending several UDP packets to very high-order ports (above 30,000), the idea being that it is very unlikely that there is an application listening on these ports. Therefore, when the destination machine receives the packets, it sends back an ICMP "port unreachable" packet. Traceroute need simply distinguish between the two types of ICMP error messages.

Recall that the route between any two machines is not fixed and in fact can change even between fragments of a single packet. How can traceroute provide

any useful information in this kind of dynamic environment? The answer is that although routes do change, they change slowly (relative to packet transit times) and sporadically. Thus, from the perspective of traceroute, the route between two hosts is quite stable.

The usage for traceroute is

traceroute destination_host

One can use either the IP address or hostname (in which case a DNS lookup is made to map the name to an IP address; see Section 1.8).

The user can specify the starting port number for the UDP packets with the -p option. The default (in Red Hat 6.1) is 33434, which is useful to know if one is trying to recognize traceroute traffic.

If all goes well, each router sends back a "time exceeded" packet as the TTL reaches 0. Some routers will actually forward a packet with a TTL of 0 (this is a bug), and some will either not send a "time exceeded" packet or choose an initial TTL that is too small to reach the sending machine. Other interesting results are possible (see the traceroute man pages for some examples), so interpretation of the results can require some effort.

Some useful options for traceroute are:

- **-f ttl** Set the initial time-to-live value for the first outgoing packet to "ttl."
- **-I** Use ICMP ECHO instead of UDP datagrams.
- **-m max-ttl** Set the maximum number of hops used in outgoing packets to "max-ttl."
- **-p port** Set the base UDP port number used in probes. As mentioned earlier, the default value is 33434.
- **-v** Provide more verbose output.
- **-w** time Set the time to wait for a response to "time."

More information on traceroute can be found in the man page.

1.9.6 tcpshow

The output of tcpdump, as seen in the preceding examples and in Chapter 4, can be quite terse and require some experience to tease out the information. The tcpshow program is designed to provide a more human-readable format for the packets.

Consider the following tcpdump trace:

```
08:00:03.760332 10.130.219.103.www > 10.10.205.136.1063:
. 2850371889:2850373345(1456) ack 2835338 win 18928 (DF)
```

Now, consider the following result from tcpshow:

```
Packet 1
Timestamp:              05:01:03.760332
Source Ethernet Address:      01:EA:1E:23:66:11
Destination Ethernet Address: 10:6A:83:41:64:60
Encapsulated Protocol:      IP
IP Header
Version:              4
Header Length: 20 bytes
Service Type: 0x00
Datagram Length: 1496 bytes
Identification: 0xA57A
Flags: MF=off, DF=on
Fragment Offset: 0
TTL: 248
Encapsulated Protocol: TCP
Header Checksum: 0x1A29
Source IP Address: 10.130.219.103
Destination IP Address: 10.10.205.136
TCP Header
Source Port: 80 (www)
Destination Port: 1063 (<unknown>)
Sequence Number: 2850371889
Acknowledgement Number: 0002835338
Header Length: 20 bytes (data=1456)
Flags: URG=off, ACK=on,  PSH=off
RST=off, SYN=off, FIN=off
Window Advertisement: 18928 bytes
Checksum: 0x4E68
Urgent Pointer: 0
TCP Data
..............
<*** Rest of data missing from packet dump ***>
```

The information is in a much more human-readable format. There is also more information in this output than in the tcpdump version, although all the information in this display is available from tcpdump (in fact, tcpshow calls tcpdump to obtain the information). A naive user may find the tcpshow format much more accessible, whereas a more sophisticated user may find the terse, one or two line output of tcpdump to be preferable.

Since tcpshow calls tcpdump, any changes in the output of tcpdump can cause tcpshow to fail to recognize the packets. In particular, one version of tcpdump places an extra "<" in its output. This confuses the version of tcpshow that I have. However, since I have the source code to tcpshow it was a simple matter to make the change to handle this case. This is one of the reasons I am very much a fan of free software.

1.9.7 snort

snort is billed as a "lightweight network intrusion detection system." What this means is that it is suited to intrusion detection on a single host or small network but is not designed to protect large networks. It is a sniffer like tcpdump, with added capabilities for content analysis and an expanded filtering capability. Like tcpdump, snort can log the packets in binary tcpdump format or provide an ascii report. In addition to providing the packets that pass the filter, snort can provide information about why the packet was flagged by the filter. Currently, snort only analyzes three protocols: TCP, UDP, and ICMP.

A subset of the command-line arguments for snort are:

- **-A alert** Turn alert mode on or off. In full mode, snort prints the full alerts to the alert file. In fast mode, terse output consisting of the timestamp, message, IPs, and ports is generated. If "alert" is "none" alerting is turned off.
- **-b** Log the packets in binary (tcpdump) format.
- **-c cfile** Use the configuration (rules) file "cfile."
- **-d** Dump the application layer data.
- **-F tfile** Use the tcpdump filter file "tfile." This is useful for using SHADOW filters with snort.
- **-h IP** Set the home network to "IP." This must be an IP address, not a domain name (for example, 10.10.1.0). This tells snort which packets are incoming and which are outgoing and adjusts the output to display this information.
- **-i if** Use the network interface "if."
- **-l dir** Log the packets in the directory "dir." The packets from a given IP address will be placed in a subdirectory corresponding to the IP address.
- **-N** Turn off logging. Only alerts will be processed.
- **-o** The normal order for applying rules is Alert->Pass->Log. This changes the order to Pass->Alert->Log.
- **-O** Obfuscate the IP addresses. The IP addresses are modified to hide their values; they are printed as "xxx.xxx.xxx.xxx". If the -h flag is set, only the home IP addresses are obfuscated.
- **-p** Do not go into promiscuous mode.
- **-r tfile** Read the tcpdump-generated file "tfile" instead of a network interface.
- **-s** Log the alerts to the syslog.
- **-v** Verbose output to the console. This can be quite slow.
- **-V** Show version number and exit.

- **-?** Show usage summary and exit. Remember to escape the question mark if necessary as appropriate for your shell.

As with tcpdump, snort will take the filter commands on the command line, but for anything but the simplest filter it is best to put these in a file. First let us look at a simple example of snort usage.

snort -dv -l ./log -h 10.10.1.0/24 -c snort.rules

will log the packets to the "log" directory (which must exist), displaying the application layer data as well as the header, with the class C network above as the home network. The rules file, or filter, "snort.rules" is used to determine which packets to log.

The source code for snort, as well as executable versions for some operating systems, can be found at

http://www.snort.org

or at

http://packetstorm.securify.com/sniffers/

snort rules are quite a bit more flexible than tcpdump filters and as a result somewhat more complicated. We will look at them briefly here, but for more information the documentation at the Web site should be consulted.

A rule is divided into two sections, a header and options. The header contains the rule action, protocol, and source and destination information. The options section describes the constraints on the header or content fields that will trigger the rule. There are three actions that can be taken as the result of a rule:

- **alert** Generate an alert and log the packet.
- **log** Log the packet.
- **pass** Ignore the packet.

Some examples will help to illustrate the power of snort. These are taken (with slight modifications) from the snort documentation.

- **log udp any any -> 192.168.1.0/24 :1024**
  Log UDP traffic from any IP address and any port to the class C network 192.1.x with destination port less than or equal to 1024.
- **log tcp any :1024 -> 192.168.1.0/24 500:**
  Log any packet with a source port less than or equal to 1024 and a destination port greater than or equal to 500.
- **alert any any -> 192.168.1.0/24 any (flags: SF; msg: "Possible SYN FIN scan";)**

Generate an alert for packets with the SYN and FIN flags set. The alert message will indicate that there may be a SYN FIN scan in progress.

- **alert tcp any any -> 192.168.1.0/24 80 (content: "cgi-bin/phf" offset: 3; depth: 22; msg: "CGI-PHF access";)**
  This illustrates content matching. Web traffic that contains "cgi-bin/phf" in the first 22 bytes of the content indicates an attempted PHF attack. The cgi program phf is known to have a vulnerability that allows the attacker to gain access to files (such as /etc/passwd) that are otherwise denied to them (see page 164).

There are currently 15 rule option keywords. These are

- **ack** Test the TCP acknowledgment field for a specific value.
- **content** Search for a pattern in the packet's payload.
- **depth** Modifier for the content option, sets the maximum search depth for a pattern match attempt.
- **dsize** Test the packet's payload size against a value.
- **flags** Test the TCP flags for certain values.
- **icmp_id** Test the ICMP ECHO ID field against a specific value.
- **icmp_seq** Test the ICMP ECHO sequence number against a specific value.
- **icode** Test the ICMP code field against a specific value.
- **id test** The IP header's fragment ID field for a specific value.
- **ipoption** Watch the IP option fields for specific codes.
- **itype** Test the ICMP type field against a specific value.
- **logto** Log the packet to a user-specified filename instead of the standard output file.
- **msg** Print a message in alerts and packet logs.
- **nocase** Match the preceding content string with case insensitivity.
- **offset** Modifier for the content option, sets the offset to begin attempting a pattern match.
- **resp** Active response (knock down connections, etc.).
- **rpc** Watch RPC services for specific application/procedure calls.
- **seq** Test the TCP sequence number field for a specific value.
- **session** Dumps the application layer information for a given session.

- **ttl** Test the IP header's TTL field value.

The preceding illustrates several powerful properties of snort that make it superior to tcpdump:

1. The ability to generate alerts to the syslog, console, or other logging mechanism.
2. The ability to scan the content for attack patterns.
3. The ability to add a message to the packet, indicating the reason the packet was logged or providing the message for the alert.
4. The ability to respond to an attack.

Another strength of snort is the ability to add plug-ins. A plug-in is a program which extends the abilities of a piece of software. These are familiar to the users of Web browsers, where plug-ins allow the browser to expand the types of files it can process or adds functionality that is otherwise missing. In snort plug-ins add capabilities such as the collection of statistics, storing output in a database, or special visualization tools. This extensibility makes snort a very useful tool.

1.9.8 ifconfig

The ifconfig utility is used to configure the network interfaces but can also provide information about them. It provides a variety of information about whether the interface is up and how it is configured. This gives a quick look at the different interfaces that are operating and can be used to configure the interface, for example to take it out of promiscuous mode.

Executing ifconfig with no arguments displays the status of the active interfaces. With the "-a" flag it will provide information about all interfaces, even those that are inactive. Otherwise, the arguments are used to configure the interface. It is not recommended that you play with this if you don't know what you are doing (although any damage you do can (probably) be repaired by rebooting the computer). See the man page for more information.

1.9.9 netstat

The netstat utility provides a great deal of useful information about network connections. Calling netstat with no options shows all the open sockets, which shows all the network connections as well as all the programs using sockets, such as X Windows, etc.

The information available from netstat is generally far more than one wants, so some kind of filtering is required. For example, on my home machine, the command

```
netstat -a | grep www
```

results in the line

tcp 0 0 localhost.localdoma:www *:* LISTEN

This tells me that my Web server is available only to my local host and not to any outside machine.

A few of the useful flags are:

- **-r** Show the kernel routing table. This is equivalent to "route -e".
- **-i [iface]** Show a table of all the networking interfaces. If "iface" is given, then show that particular interface. (The Unix convention of using a "[]" to indicate an optional argument is used here.)
- **-n** Do not try to resolve IP addresses into host names but rather print the address (similarly for port or user names). This can make the program run much faster.
- **-p** Display the process name and PID for the owner for each socket that is dumped.
- **-l** Display the sockets that are listening.
- **-c** Run netstat continuously.
- **-s** Display networking statistics.

As with many Linux utilities, netstat will accept a "help" flag and return the usage information.

On my home computer, which is only connected to the Internet using PPP across a modem (and hence does very little networking most days),

```
netstat -s
```

produced

```
Ip:
    7214 total packets received
    1 with invalid headers
    0 forwarded
    0 incoming packets discarded
    207 incoming packets delivered
    6798 requests sent out
Icmp:
    45 ICMP messages received
    0 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 45
    75 ICMP messages sent
    0 ICMP messages failed
```

```
    ICMP output histogram:
        destination unreachable: 75
Tcp:
    431 active connections openings
    0 passive connection openings
    0 failed connection attempts
    0 connection resets received
    0 connections established
    6961 segments received
    6563 segments sent out
    408 segments retransmitted
    58 bad segments received.
    419 resets sent
Udp:
    132 packets received
    75 packets to unknown port received
    0 packet receive errors
    223 packets sent
TcpExt:
```

As you can see, very little activity is represented by this, testifying to the fact that I have been writing this section rather than surfing the Internet. The packet counts are displayed according to the protocols, broken down into incoming and outgoing packets, with statistics about the quality of the connection implicit in the errors reported.

Using

```
netstat -l
```

will indicate which ports are listening, which is a good place to start looking for trojan programs (Chapter 7).

1.9.10 pppstats

A related utility is pppstats, which provides information about a PPP connection. PPP, or Point-to-Point Protocol, is the protocol used for most serial communication, such as that through a modem. An example output for pppstats is shown in Table 1.4.

Table 1.4 Output from pppstats.

IN	PACK	VJCOMP	VJUNC	VJERR
1327	19	0	9	0
OUT	PACK	VJCOMP	VJUNC	NON-VJ
916	18	1	7	10

This is what one might see right after initializing a PPP connection. The fields displayed are:

IN The number of bytes received by the PPP interface.

PACK The number of packets received by the PPP interface.

VJCOMP The number of compressed TCP packets received by the PPP interface.

VJUNC The number of uncompressed TCP packets received by the PPP interface.

VJERR The number of corrupted compressed TCP packets received by the PPP interface.

OUT The number of bytes transmitted by the PPP interface.

PACK The number of packets transmitted by the PPP interface.

VJCOMP The number of compressed TCP packets transmitted by the PPP interface.

VJUNC The number of uncompressed TCP packets transmitted by the PPP interface.

NON-VJ The number of non-TCP packets transmitted by the PPP interface.

More information can be obtained using the "-v" flag on the command line. See the man page for the details.

1.9.11 lsof

A final useful utility is lsof, which stands for "list open files." This will list all the open files belonging to all the active processes on the machine. This is extremely useful for determining who is looking at what (for instance, your syslog file) and what programs are dependent on which files (which can be useful for determining which files to protect from trojans).

An open file may be a regular file (for example, a text file you are editing), a directory, a library, a stream or network file (socket or NFS file), or various kinds of "special" files.

The lsof program comes with many Unix implementations and can be obtained at:

ftp://vic.cc.purdue.edu/pub/tools/unix/lsof

Some of the useful command line options for lsof are:

- **-i [spec]** List Internet files. If "spec" is provided, it lists those files whose Internet address matches "spec." For example, on my home system (which at

the time was not connected to any network), the command "lsof -i" produced the output in Table 1.5. Each entry had a TYPE=IPv4 and NODE=TCP, which were removed in the interest of space. It may come as a surprise to some that there are TCP ports active even though the machine is not connected to a network.

FD stands for file descriptor. In the listing in Table 1.5, the number corresponds to the file descriptor number of the file, while the "u" indicates that it is open for both reading and writing. An "r" or "w" would indicate read or write access, respectively.

The Internet address is specified in the following format:

[protocol] [@hostname | hostaddr] [:service | port]

The Unix convention of square brackets "[]" is used to indicate optional arguments and "|" to represent "or." Here,

- **protocol** is a protocol name (TCP or UDP).
- **hostname** is an Internet host name.
- **hostaddr** is an IP address.
- **service** is the name of a service (for example, smtp). See /etc/services for a list of these names.
- **port** is a port number.

Both service and port can be a comma-delimited list. Thus, the command

lsof -i :1590

produces the listing for the single application (gnomepage) that is listening on port 1590.

Table 1.5 An example of the output from "lsof -i," listing all the open Internet files.

COMMAND	PID	USER	FD	DEVICE	NAME
gnome-ses	13587	dmarche	3u	37496	*:1578 (LISTEN)
magicdev	13612	dmarche	6u	37603	*:1582 (LISTEN)
panel	13629	dmarche	6u	37714	*:1587 (LISTEN)
gnome-nam	13633	dmarche	4u	37680	*:1586 (LISTEN)
gmc	13639	dmarche	6u	37769	*:1588 (LISTEN)
gnomepage	13669	dmarche	5u	38094	*:1590 (LISTEN)
gen_util_	13671	dmarche	5u	38082	*:1589 (LISTEN)
rp3	13675	dmarche	5u	38146	*:1592 (LISTEN)
rp3	13677	dmarche	5u	38132	*:1591 (LISTEN)

- **-n** Inhibit the conversion of network numbers to host names. This conversion can take quite a bit of time, so it is a good idea to suppress it for most applications.
- **-o | -s** Toggle between showing the file size and the file offset. Only one of these flags may be used.
- **-P** Like the "-n" option, this suppresses the conversion of port numbers to port names.
- **+|-r [t]** Put lsof in repeat mode. This causes the program to run every "t" seconds. If the "-" is used, lsof will run forever (until killed), whereas with the "+" lsof will exit on the first iteration in which no open files are listed.
- **-R** List the process ID (PID) of the parent process under the PPID heading.
- **-t** Terse output. For example, in the example depicted in Table 1.5, running lsof -i -t will return just the list of PID numbers. This is particularly useful for piping the output to the "kill" program to kill off all processes that have a particular file open.
- **-U** List Unix socket files.
- **-v** Print the version information.
- **-V** List the items that lsof was asked to find but could not.

We have seen some examples of the use of lsof previously. Let us look at a few more. To list the open files for user dmarche, use

lsof -u dmarche

To find the processes that have opened the file "foo," use

lsof foo

It is interesting to try this with the text editor "vi." First type

vi foo

in a window. In another window, type

lsof foo

What happened? On my machine, the lsof command returned nothing. How could this be? I am clearly editing the file foo. To investigate further, try

lsof -u <yourusername> | grep foo

You will see something like

vi 18370 dmarche 4u REG 3,7 98304 1884896 .foo.swp

What has happened is that when vi is started, it opens a temporary file, ".foo.swp" in this case, and copies your file into it. Once this is done, the original file is closed until you tell vi to write the changes. Thus, lsof was not wrong in stating that no processes had the file foo open. The problem was that you asked the wrong question. Asking the right question is always a goal worth striving for.

See the man pages for a more complete listing of the options and for many more examples.

It should be noted that on some systems lsof will not show any files that are not opened by processes owned by the user. Thus, many of the preceding commands will not work (actually, they will work, but their output will not be complete). If you compile lsof and want this security feature, use the compile-time option HASSECURITY. On my version of Red Hat Linux 6.1, lsof came installed in "unsecure" mode, meaning that any user can obtain information about all open files for all users. There are two ways to tell whether your version is installed in this manner (assuming you did not do the installation yourself): you can run it and see if you see processes that do not belong to you (for example, root processes), or you can use the -h flag, which will give various information about the program, as well as say

Anyone can list all files;

if the program is "unsecure."

This is a tiny example of the potential utility of lsof. We will return to this program in Section 5.6.3, where we consider its implications for host-based security.

1.10 FURTHER READING

There are many books on networking and TCP/IP, and I will refrain from listing them here. As mentioned earlier, a very good place to start is Stevens [1994]. Other books include Comer [1991], Loshin [1997], and Simoneau [1997].

A book that focuses specifically on IP and the three protocols we have discussed in this chapter is Hall [2000]. O'Reilly has a number of books on TCP/IP, networking, DNS, and related topics, and these tend to be quite useful references.

For information on available utilities, the definitive reference is always the man page. However, unless you know the name of the command you want to reference, it is difficult to find it in the man pages. There are several books that provide manual pages for a given operating system, one of which, for Linux, is Petron [2000].

The definitive references for the Internet protocols are the RFCs. These lay out the details for each protocol and address the requirements and options available. It can be tedious looking through these, particularly as they are not indexed. Two books that help with this are Loshin [2000b] and Loshin [2000a].

2

Network Statistics

2.1 INTRODUCTION

This chapter looks at some issues related to collecting, measuring, and analyzing network traffic. This will be a brief introduction aimed at introducing some of the issues involved, with a focus on applications of statistical methods to the problems. Some suggestions for further reading are provided at the end.

There are a number of issues relevant to network traffic modeling. First, we will look at some work on estimating network intensities, such as transit times. This is a statistical treatment, looking at determining good ways to collect the data. Then we will investigate some work on network tomography, which involves inferring intensities on routes from information at the endpoints.

The next section involves issues related to modeling the distributions of network traffic. It turns out that network traffic has very interesting structure and is much more complex than simple Poisson models might lead one to believe.

A brief look at projects undertaking the mapping of the Internet will be followed by some discussion on visualization techniques for network data. The final section will provide pointers to further reading.

2.2 NETWORK TRAFFIC INTENSITIES

In this section, we will look at two issues involved in the collection and analysis of network intensity data. The first is the question of how we collect data given constraints such as the number of sensors we can place or restrictions on how much impact we wish our collection to have on the network. The second section

will look at a specific estimation problem, the so-called "network tomography" problem.

2.2.1 Design of Experiments

Consider the problem of monitoring a network for such quantities as transit times between nodes, delays at a node, and so forth. We wish to collect data to estimate the quantity of interest. Following Fedorov and Flanagan [1998], this section will investigate the problem of designing an experiment to estimate a single quantity.

One of the constraints that we wish to impose is that data can be collected at only a small number of nodes. There are two models for data collection. If we have a sensor at each node for which data are taken, we call this a "passive" collection, or a "passive sensor." If the data are taken by sending packets out and measuring responses to the packets, this is called "active" collection, or an "active sensor." Resource constraints may force us to use a small number of passive sensors rather than have a sensor on every node. If an active sensor is used, we may wish to minimize the impact of the collection on the network.

We will consider round-trip transit time as measured by the ping program (Section 1.9.1). As in Fedorov and Flanagan [1998], we will restrict our discussion to the problem of determining the transit time between a single host and S other machines. We will assume N, the number of observations to be made, and S are given. The question then is how best to allocate our measurements in order to get the best estimate of transit time.

We will use the notation of Fedorov and Flanagan [1998] throughout. The S hosts will be denoted $X = (x_1, ..., x_S)$. Let our variable of interest be denoted $U = (u(x_1), ..., u(x_S))^T$, and an observation is

$$y_j = u(x_i) + \varepsilon_j(x_i). \tag{2.1}$$

This is the jth observation at the ith node. At node i there will be r_i observations taken. We assume that no changes in the value we are estimating occur during the collection of the observations. The errors $\varepsilon_j(x_i)$ are assumed to have zero mean and to be uncorrelated with variance σ^2, independent of the node. Let

$$K = E[(U - E[U])(U - E[U])^T] \tag{2.2}$$

be the covariance matrix.

The goal is to design a data collection experiment. We wish to determine which N nodes to sample to best predict the vector U. This set of nodes is called the experimental design. The predictor will assign a weight to each node, and the prediction will be a weighted sum of the observations. Thus, we are considering a linear predictor in this work.

The experimental design is defined to be

$$\xi_n = \{p_i, x_i\}_1^n, \quad p_i = r_i/N, \quad N = \sum_{i=1}^{n} r_i, \quad x_i \in X, \quad n \leq S. \tag{2.3}$$

The p_i denote the weight on the node or the proportion of the observations to be taken at node i. The x_i are the design points of the experiment, the nodes at which the observations are to be made.

Upon collection of our observations, we average them to obtain

$$Y(\xi_n) = \begin{pmatrix} \frac{1}{r_1}\sum_{j=1}^{r_1} y_j(x_1) \\ \vdots \\ \frac{1}{r_n}\sum_{j=1}^{r_n} y_j(x_n) \end{pmatrix} = \begin{pmatrix} Y_1 \\ \vdots \\ Y_n \end{pmatrix}. \tag{2.4}$$

Thus, we collect r_i observations from each of the nodes defined in ξ_n. We average these observations for each node. This is the observation vector that will be used to make the predictions. The averaging could be thought of as a method for denoising, or reducing the variance, of the estimate. Instead of averaging, one could consider a robust estimate of location, such as the median, but we will not consider this here.

Note that one way to reduce the impact on the network would be to make our measurements during off hours. However, if the purpose is to measure delay, this would defeat the purpose. For some quantities, we have no choice but to make our measurements during peak hours.

It is important to keep in mind that we are predicting the activity across the entire collection of nodes from measurements collected at a subset of nodes. As mentioned earlier, this is often necessary due to lack of resources or a desire to reduce the impact of the data collection on the network. We want to design our experiment (select the monitored nodes) in such a way that our estimate is as accurate as possible. To this end we need a way to measure the accuracy of our estimate.

Given $\hat{U}$ an estimate of U, define the matrix of expected squared residuals $D(\xi_n, \hat{U})$ as

$$D(\xi_n, \hat{U}) = E[(\hat{U} - U)(\hat{U} - U)^T]. \tag{2.5}$$

We write $D(\xi_n, \hat{U}) = D(\xi_n)$ when the estimator is clear.

Let $K(\xi_n)$ be the submatrix of K corresponding to the nodes $x_1, ..., x_n$, let $K(x, \xi_n)$ be the column of covariances between $u(x)$ and the $u(x_1), ..., u(x_n)$, and let $K(Z, \xi_n)$ be the corresponding matrix for the nodes $Z \subset X$. Finally, let $W(\xi_n)$ be the diagonal matrix with elements $r_i\sigma^{-2}$. Fedorov and Flanagan show that the best estimator (in the sense of minimizing the expected squared residuals $D(\xi_n, \hat{U}(Z))$) is

$$\hat{U}(Z) = K^T(Z, \xi_n)(K(\xi_n) + W^{-1}(\xi_n))^{-1}Y(\xi_n). \tag{2.6}$$

This looks complicated, but it is nothing more than a linear combination of the Y's.

Often, one wishes to optimize some function of $D(\xi_n)$, such as the trace or maximal diagonal element. This, in general, is not possible in closed form, and Fedorov and Flanagan give some approximations with an optimization algorithm. They then proceed to illustrate the results with an example of estimating round-trip times to a number of sites.

The algorithm is as follows. First, we are looking for the design that minimizes the function $\log(|D(\xi)|)$. Such a design is called "D-optimal". We have the equality

$$D(\xi) = (K^{-1} + W(\xi))^{-1} \tag{2.7}$$

when $N = S$ and $Z = X$. To choose a given number m of nodes from a design ξ, we will choose the m largest weights from W, setting the rest to 0. Now, set $\alpha_0 = 1/N$, $p_{0i} = \alpha_0$, which defines the initial design ξ_0. Let γ be a small positive number less than 1 and $\epsilon > 0$ a small number (used as a stopping criterion).

1. Given ξ_t and D_t, find the index of the largest element of the diagonal of D_t and increase the corresponding weight by α_t.

2. Using the new weights, construct a new D matrix and find the index of the smallest nonzero diagonal element of this matrix. Reduce the corresponding weight by α_t. This produces the new design ξ_{t+1}.

3. If $|D(\xi_{t+1})|/|D(\xi_t)| < 1 - \gamma$ (that is, the change in D is "large"), set $\alpha_{t+1} = \alpha_t$ and go to step 1. Otherwise, set $\alpha_{t+1} = \alpha_t/2$. If $\alpha_{t+1} < \epsilon$, return ξ_{t+1}; otherwise, go to step 1.

In their experiment, Fedorov and Flanagan used ping to determine the round-trip time between a single node and 39 sites. They used only the ten largest weights in the design; hence only ten sites were to be used in the estimate. The number of pings sent to a site was proportional to the weight. They report that the D-optimal design is nearly four times as efficient as the uniform design (using all 39 sites) in their experiment. This may at first seem counterintuitive since it would seem that using all the sites should provide more information than using only ten, but remember that the total number of observations is fixed. The gain comes from the fact that multiple observations from a site can be used to reduce the variance of the estimator through averaging. This is the "denoising" referred to previously.

All of this assumes knowledge of the covariance matrix K. This is estimated prior to the experiment. It must be assumed that K is stationary throughout the data collection and that the estimate is of good quality. Fedorov and Flanagan admit that they used a simple approach for this estimate and that more care should be taken.

2.2.2 Network Tomography

A related problem is described in Vardi [1996] and Tebaldi and West [1998]. Called "network tomography" by Vardi, the idea is to infer the traffic intensity across the routes in a network using only measurements of intensity at the nodes.

We start with a network of nodes with directed connections between the nodes. In graph theory, this is called a directed graph, or digraph. Traffic can travel from one node to another if and only if there is a connection from the first to the second. This is illustrated in Figure 2.1. In this figure, there are two possible routes from a to b: $a \to b$ and $a \to c \to b$ (while the route $a \to c \to d \to c \to b$ is theoretically

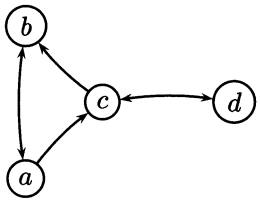

Fig. 2.1 A simple four-node network with directed arcs indicating links between nodes.

possible, we do not allow traffic to retrace its steps in this model). There is a single route possible from b to c: $b \to a \to c$.

Assume a network of n nodes. Then there are $c = n(n-1)$ source/destination (SD) pairs. Let r denote the number of directed links. In Figure 2.1, $n = 4$, $c = 12$, and $r = 6$.

The work proceeds along two lines. First, in the simpler case, the routes are fixed and known. In the second case, the routes are random but the transition probabilities between nodes are known. We will consider the fixed route in this section.

In the first case, define the routing matrix A as an $r \times c$ binary matrix, where $a_{ij} = 1$ if and only if the ith link is in the route between the jth SD pair. This is illustrated in Table 2.1. Note that although it is possible to go between a and b via the node c, the routing matrix indicates that this route is not allowed (see the first column of the routing matrix). Only the direct link from $a \to b$ is allowed in the SD pair ab, so the routing matrix provides the information necessary to decide the route between any two nodes, and these routes are unique.

For a set of measurement periods, we will measure the traffic along each directed link. These observations are denoted $Y^{(k)} = (Y^{(k)_1}, ..., Y_r^{(k)})$, taken at times $k = 1, ..., K$. The underlying random variables are $X^{(k)} = (X_1^{(k)}, ..., X_c^{(k)})$,

Table 2.1 A routing matrix for routes for the network in Figure 2.1. The columns correspond to source/destination (SD) pairs, while the rows correspond to the links between nodes. Thus, a 1 in position i, j indicates that link i is used in the route corresponding to source/destination (SD) pair j. A blank corresponds to a 0, indicating that the link does not appear in the route.

	ab	ac	ad	ba	bc	bd	ca	cb	cd	da	db	dc
$a \to b$	1											
$a \to c$		1	1		1	1						
$b \to a$				1	1	1	1			1		
$c \to b$							1	1		1	1	
$c \to d$			1			1			1			
$d \to c$										1	1	1

where each $X_j^{(k)}$ is the number of transmitted messages for the SD pair j at measurement period k. Thus,

$$Y^{(k)} = X^{(k)}A. \tag{2.8}$$

We assume that the $X_j^{(k)}$ are independent for each j and k, and distributed as a Poisson distribution, with a different rate for each SD pair. In other words, $X_j^{(k)} \sim Poisson(\lambda_j)$, where the Poisson distribution is defined as

$$f(x;\lambda) = \frac{\lambda^x e^{-\lambda}}{x!}, \tag{2.9}$$

where x is constrained to be a nonnegative integer. The mean and variance of a Poisson random variable with density as in Equation (2.9) are both λ. For more information about the Poisson distribution, see any statistics text, such as Hogg and Craig [1995].

Thus, the goal is to estimate $\lambda = (\lambda_1, ..., \lambda_c)$ from the observations $Y^{(k)}$. First we must determine whether we can in fact estimate λ.

Recall that a parameter vector is identifiable if it can be determined uniquely. For example, if we can only determine $\lambda_1 + \lambda_2$, we cannot "identify" the vector (λ_1, λ_2). For another example of nonidentifiability, consider the problem of trying to identify a mixture of two uniform densities. Consider Figure 2.2. This could be modeled as a mixture of uniform densities in many ways, in particular

$$\frac{1}{4}U\left(0, \frac{1}{2}\right) + \frac{3}{4}U\left(\frac{1}{2}, 1\right), \tag{2.10}$$

$$\frac{1}{2}U\left(0, 1\right) + \frac{1}{2}U\left(\frac{1}{2}, 1\right). \tag{2.11}$$

There is no way to distinguish the parameters in these models since the different parameters produce the same density.

In the Poisson model we are considering here, λ is only identifiable when all the columns of A contain at least one nonzero entry and are distinct. Clearly, real-life routing matrices would generally not have a column of zeros (this corresponds to a source/destination pair that has no route: "you can't get there from here"). An example of identical columns can be found by considering Figure 2.1 and allowing the following two routes:

$$ab : a \to c \to b \to a \to b$$

and

$$ac : a \to b \to a \to c \to b \to a \to c,$$

which correspond to two columns equal to $(111100)^T$. This also is clearly not likely in a real routing matrix due to the number of backtracks. Clearly, these routes are not possible in real life since they both would stop as soon as the destination node had been reached.

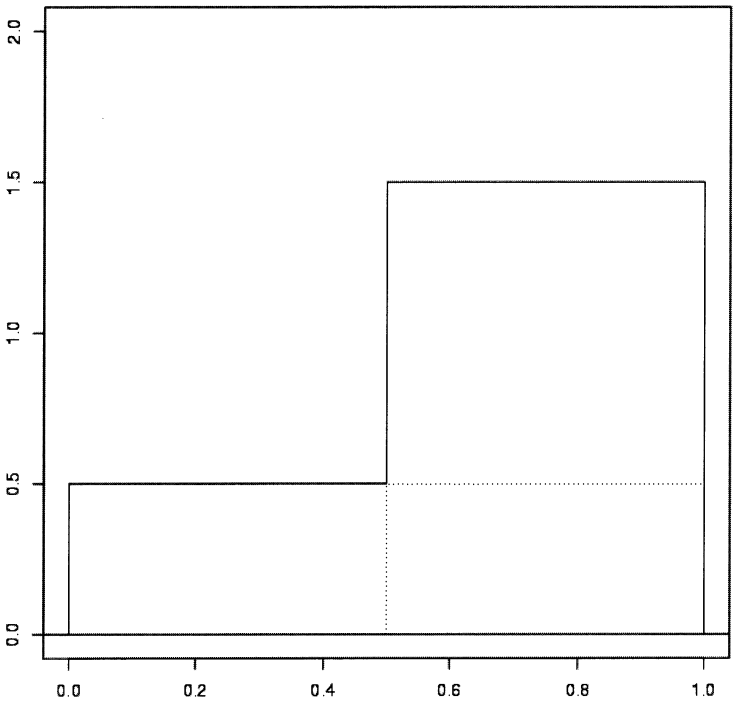

Fig. 2.2 A mixture of uniform densities. The dotted lines indicate two different ways of representing the mixture, as indicated by Equations (2.10) and (2.11).

It should be noted that there are situations in which a routing table may indicate no route to a host. For example, with portsentry (Section 5.6.5), one has the option of "dropping the route" to an attacking host, which means that your machine will simply refuse to send any packets to the attacker, making it difficult to execute a successful network attack on your machine. Normally, however, there will always be a route between any SD pair.

There are several issues that are not addressed in the Vardi [1996] paper. As we saw in Section 1.2, we could in fact measure traffic between SD pairs directly at each node. In fact, we could make these measurements without knowledge of the routing matrix; that is, at each node, we could measure the traffic between any SD pair as it passes through the node. This allows us to infer the routing matrix, or in the case of random routing, we could estimate the transition probabilities.

This brings up an interesting question, which is more in keeping with the title "network tomography". From the traffic at a subset of the nodes, can we infer the network topology? For example, in Figure 2.1, given traffic measurements at nodes a, b, and d, can we infer the existence of node c? This is easy if c is one of the SD pairs and we are using a sniffer such as tcpdump. Once we see a packet destined for c, we know of c's existence, assuming the destination address has not been spoofed (that is, assuming that the packet is a legitimate packet destined

for a legitimate machine). It is somewhat more difficult if we are only measuring traffic intensities at a, b, and d. A harder problem would be if c is a router, with no traffic specifically destined to or from it. For example, if we were sampling TCP traffic, we might not see any traffic to or from c. On the other hand, if we were measuring ICMP traffic, we might very well see traffic, and if we were allowed to inject traffic (for example, through traceroute) we could in effect probe for c directly. This is the problem addressed by the people trying to map the Internet (Section 2.4). A much more difficult problem would be to infer the existence of c and estimate the routing matrix from TCP traffic alone using, for example, delays between the handshaking to infer the existence and number of routers between any SD pair. Inferring the existence of c from traffic intensities at the other three nodes alone would also be a difficult task that would be worth considering. Under what constraints on the network topology can these questions, and variations, be answered? This is an interesting set of open questions.

Returning to the "easier" question, we need to estimate λ from the $Y^{(k)}$. To accomplish this, we use the EM algorithm.

"EM" stands for expectation/maximization. The EM algorithm is a method for maximum likelihood estimation in missing data problems. To illustrate the idea, consider the problem of fitting a mixture of two normals to data. The probability density function (PDF) is

$$f(x) = p\phi(x, \mu_1, \sigma_1^2) + (1-p)\phi(x, \mu_2, \sigma_2^2), \tag{2.12}$$

where ϕ is the univariate normal density

$$\phi(x, \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}. \tag{2.13}$$

Assume further that we knew the means and variances and were only interested in determining the value of p. We have data, $x_1, \cdots, x_n$, which are assumed to have been drawn from the distribution f, so each of the x_i "came from" one of the two components. If, for each observation, we knew which of the components was the source of the observation, the estimation of p would be easy: $\hat{p} = n_1/n$, where n_1 is the number of observations from component 1. Unfortunately, this information is unavailable, or "missing."

The EM algorithm approaches this problem by first estimating the missing data (the "expectation" part), and then, given the estimated information, we obtain the parameter that best fits with this estimate and the data (the "maximization" part).

In our example, we start with some guess for p, say $p = 1/2$. For each observation, x_i, we compute the posterior probability (the probability that the observation was drawn from component 1) and then use this to obtain a new estimate for p. Thus, for iteration j (letting $p_0 = 1/2$), we have

$$\tau^{(i)} = \frac{p^{(j)}\phi(x_i, \mu_1, \sigma_1^2)}{f(x_i)}, \tag{2.14}$$

$$p^{(j+1)} = \frac{1}{n}\sum_{i=1}^{n}\tau^{(i)}. \tag{2.15}$$

This is repeated until the algorithm converges. Thus, Equation (2.14) is the "E" step and Equation (2.15) is the "M" step.

The EM algorithm is a very general procedure. One of the best references for it is McLachlan and Krishnan [1997]. The original reference is Dempster et al. [1977].

In the case at hand, we observe sums of Poisson random variables (Equation (2.8)). It would be much easier to estimate λ if we observed the random variables themselves. In a sense, we have lost information, so the EM algorithm is a natural approach to try.

The EM algorithm for λ can be derived as

$$\lambda^{t+1} = \frac{1}{K}\sum_{k=1}^{K} E[X^{(k)}|Y^{(k)}, \lambda^{(k)}], \tag{2.16}$$

for $n = 0, 1, ...$, where the superscript t has been suppressed for the $\lambda^{(k)}$ in the expectation. The expectations in the sum are conditional expectations - that is, the expected value of X conditional on the values of Y and λ. Unfortunately, this is hard to calculate, and Vardi [1996] provides several solutions to this dilemma.

The first solution relies on a normal approximation. Let Λ be the diagonal matrix whose diagonal entries are the components of λ. An approximation allows a solution to Equation (2.16). After some derivation, the EM algorithm for this approximation is shown to be

$$\lambda^{(t+1)} = \frac{1}{K}\sum_{k=1}^{K}\left[\lambda^{(t)} + \Lambda^{(t)}A^T(A\Lambda^{(t)}A^T)^{-1}(Y^{(k)} - A\lambda^{(t)})\right]. \tag{2.17}$$

Equation (2.17) is a particularly nasty looking one at first glance. It comes from the following approximation. Recall that the mean and variance of a Poisson distribution with parameter λ are both λ. Since the conditional expectation in Equation (2.16) is so hard to calculate for the Poisson distribution at hand, consider approximating the distribution of X as a normal with mean λ and covariance Λ. Then, the joint distribution of X and Y is

$$N\left(\begin{pmatrix}\lambda \\ A\lambda\end{pmatrix}, \begin{pmatrix}\lambda & \Lambda A^T \\ A\Lambda & A\Lambda A^T\end{pmatrix}\right), \tag{2.18}$$

from which the conditional distribution of $X|Y$ can be derived (see Seber [1984], pages 18–19), resulting in the formula (2.17).

The preceding normal approximation is not particularly good for small λ's, so Vardi [1996] also provides one based on approximating the distribution of the average of the Y's as a normal (which is justified by the central limit theorem if K is large).

The EM algorithm is relatively easy to apply, but Vardi prefers a technique based on moments. The idea is to use the equations for the mean ($\bar{Y}_i$) and covariance (S) of the Y_i to construct a family of equations to solve

$$\begin{pmatrix} \bar{Y} \\ S \end{pmatrix} = \begin{pmatrix} A \\ B \end{pmatrix} \Lambda, \tag{2.19}$$

where B is constructed via element-wise products of rows of A (the details can be found in Vardi [1996]). The basic point is that this is a relatively simple linear algebra problem, which can be easily solved. Set

$$\dot{a}_{.j} = \sum_i a_{ij}$$

and similarly for $\dot{b}_{.j}$, and

$$\hat{\lambda}_j(A, Y, \lambda) = \frac{\lambda_j}{\dot{a}_{.j}} \sum_i \frac{a_{ij} Y_i}{\sum_k a_{ik} \lambda_k};$$

then the solution becomes the iteration

$$\lambda_j^{t+1} = \frac{\dot{a}_{.j}}{\dot{a}_{.j} + \dot{b}_{.j}} \hat{\lambda}_j(A, Y, \lambda^t) + \frac{\dot{b}_{.j}}{\dot{a}_{.j} + \dot{b}_{.j}} \hat{\lambda}_j(B, S, \lambda^t). \tag{2.20}$$

Equation (2.20) provides an algorithm for estimating λ. It assumes that the Poisson model is correct, an assumption that we will see is not always warranted. Vardi [1996] discusses this and provides a solution that allows the user to "de-weight" the second-moment terms, which are the ones that make use of the Poisson model assumption.

In the case of random routing, the matrix A consists of transition probabilities rather than 0's and 1's. This is somewhat more complicated, and we leave this for the interested reader to pursue.

This section has developed a number of ideas for measuring and modeling network traffic. It is important to keep in mind that network traffic is complicated, requiring care in selecting models, and it typically consists of extremely large data sets. Further, the ability of the researcher to measure the data can be restricted, as is the case in many interesting problems. In the case of network data, the restrictions are often a result of security policy or the desire to limit the impact of the measurement on the network. We will delve a little deeper into modeling network traffic in the next section.

2.3 MODELING NETWORK TRAFFIC

In this section we consider the problem of describing and modeling the distributions of various statistics of network traffic. In the previous section, we looked at a particular kind of inference, determining the traffic loads between source and destination pairs. Here we are more interested in characterizing network traffic in general.

The previous work assumed that the traffic intensity was distributed as a Poisson random variable. As we will see, this assumption breaks down as we investigate network traffic in more detail.

Network traffic is quite complex. For example, consider Web traffic. When you type in a URL to your browser, many things that you are mostly unaware of happen behind the scenes. We have seen that the TCP handshaking is used to initialize the connection. Data pass back and forth until the session is closed. However, a Web session usually consists of more than just one session. Even if you only go to a single page, you will probably initiate several sessions. Each image you download is a separate session, as are other files that may be loaded by the page. After a few seconds of reviewing the page, you then select a new URL and the process starts over. In addition to these explicit sessions, there are generally DNS lookups that occur to obtain the IP addresses of the pages and images.

Thus, there are several levels to the network data. Within each TCP session there are the individual packets, whose statistics are determined primarily by the network and hardware considerations. The number of sessions spawned by the main session is different for each Web page and thus has another distribution associated with it. Finally, there is the user input, which determines the time between new sessions.

I will consider several models for network traffic. These are by no means a complete listing of the work done on this problem. I will provide references to other work, so interested readers can learn more.

In the simplest case, we consider the initiation times of TCP user sessions. A basic probability course would tell us that arrival times are Poisson, so this is a good starting place. Note that it is also reasonable to believe that the rate is slowly varying (diurnal), and in fact several groups (Paxson and Floyd [1995], Nuzman et al. [2000]) have found that (some) user session initiation times are well modeled by a Poisson distribution with a slowly time-varying rate. This model breaks down, however, if one considers other factors such as the data transfer times for FTP traffic or activity that spawns new activity, such as news and Web transfers.

It is shown in Paxson and Floyd [1995] that telnet and FTP session arrival times are well-modeled by the Poisson process (Equation (2.9)). Recall that a Poisson process is one where the counts within fixed time intervals are distributed as Poisson. Thus, when talking about continuous random variables as being "Poisson," we mean they come from a Poisson process. In the same paper, the claim is made that email, Web and news are not Poisson.

Let us look at some data. Figure 2.3 shows data collected over a two-hour period. The interarrival times between connection requests to a mail server are plotted against arrival times. Notice the burstiness of the data (gaps in the plot). The data are quite correlated (with a correlation of 0.33 in this case), as can be

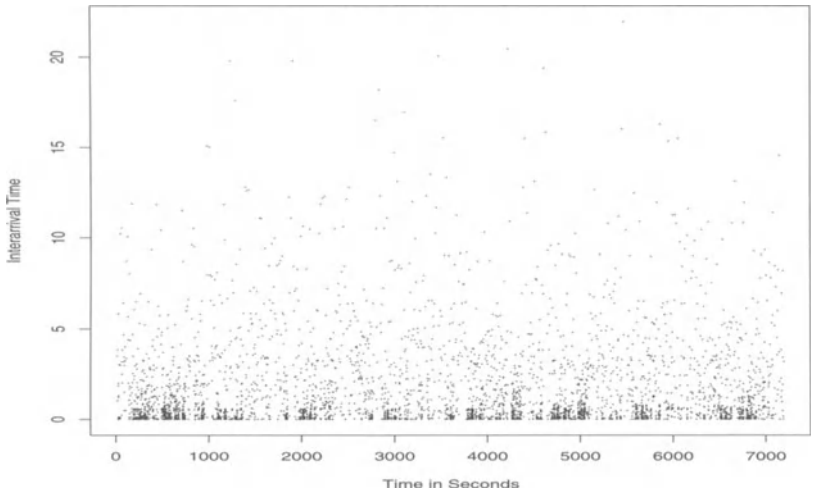

Fig. 2.3 Scatter plot of the interarrival times of connections to a mail server. The data were collected over a two hour time period.

seen in Figure 2.4. This tells us that the data are not independent - that the time between packets now is to an extent related to the time between packets in the recent past.

Figure 2.5 shows another view of these data. Here, the interarrival times between connection requests are depicted as a histogram with a gamma distribution

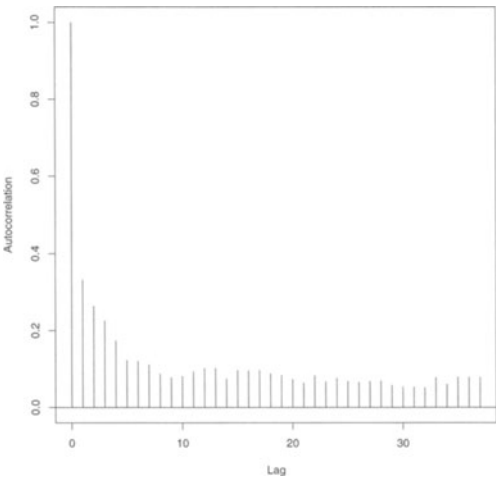

Fig. 2.4 Autocorrelation of the data in Figure 2.3.

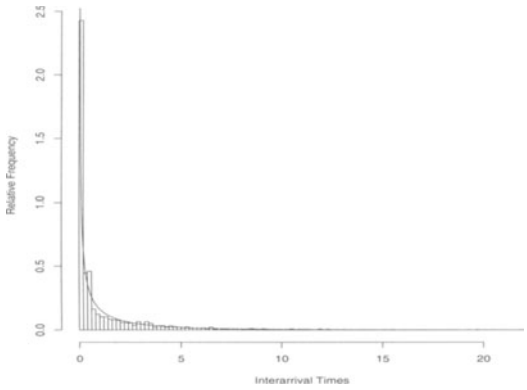

Fig. 2.5 Histogram of the interarrival times of connections to a mail server. The data were collected over a two hour time period. The times are in seconds. A gamma distribution fit to the data is shown as a curve overlaid on the histogram.

fit to the data,

$$\gamma(x;\alpha,\beta) = \frac{1}{\beta^{\alpha}\Gamma(\alpha)} x^{\alpha-1} e^{-x/\beta}. \tag{2.21}$$

The parameters of the gamma distribution fit are $\alpha = 0.278$ and $\beta = 4.241$. There are 6104 observations in these data. Figure 2.6 depicts the same graph zoomed in to the range from 0 to 5 seconds. As can be seen, the fit is not too bad, although there seems to be something interesting happening at around 1/2 second.

Paxson [1994] calculates statistics for several different quantities, such as number of bytes per session and duration of session for several different applications

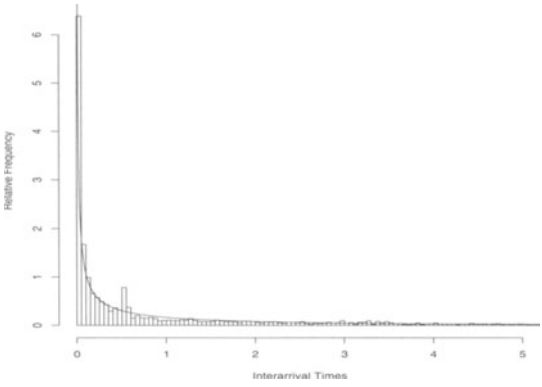

Fig. 2.6 Histogram of the interarrival times of connections to a mail server (Figure 2.5) zoomed in to the range from 0 to 5 seconds. A gamma distribution fit to the data is shown as a curve overlaid on the histogram.

(telnet, email, news, and FTP). He fits a number of distributions to the data, performing tests of fit to determine the best fit. The distributions investigated are:

$$\begin{aligned} \text{Extreme: } f(x) &= \frac{1}{\beta} e^{-\frac{x-\alpha}{\beta}} e^{-e^{-(x-\alpha)/\beta}}; \\ \text{Pareto: } f(x) &= \frac{\alpha k^{\alpha}}{x^{\alpha-1}}; \\ \text{Exponential: } f(x) &= \frac{1}{\beta} e^{-x/\beta}. \end{aligned}$$

In addition, he considers the lognormal and log-extreme distributions. A random variable is said to have a log-f distribution if $\log(x)$ has distribution f. In this case, the logarithms are taken base 2.

Using a series of tests based on the χ^2 test, a "best fit" distribution is found for various quantities. Some of these are reproduced in Table 2.2. These data represent the analysis of 3 million connections. In the table, "originator bytes" refers to the number of bytes per packet from the originator of the session and similarly for "responder bytes." In FTP, data transfers often occur in "bursts," sessions that occur less than 4 seconds apart. These can be caused by multiple gets or puts, in which several files are transferred in rapid succession. The number of bytes in these burst sessions is referred to as the "burst bytes" in the table.

The conclusion of the Paxson [1994] paper is that although the models are not perfect, they do a good job of approximating the empirical distributions of the network data investigated. Further, different quantities are distributed according to different families of distributions, and none of the quantities considered was well-modeled as a Poisson process.

As we saw in our small example (Figures 2.3–2.6), network data are complicated, even if we restrict our investigation to relatively simple problems such as session interarrival times. Several authors have investigated these issues and found interesting structure. Leland et al. [1994] were among the first to comment on the self-similar nature of network traffic. They first illustrate the self-similarity graphically by noting that the traffic looks similar at different scales. Our little 6000 point data set is not large enough to provide evidence nearly as convincing as in Leland et al. [1994]; however we can illustrate it on a small scale (Figure 2.7).

Table 2.2 Models selected in Paxson [1994] for various traffic quantities.

Protocol	Variable	Model
Telnet	Originator bytes	log-extreme
	Responder bytes	log-normal
	Duration in seconds	log-normal
NNTP	Originator bytes	log-normal
SMTP	Originator bytes	log-normal
FTP	Connection bytes	log-normal
	Session bytes	log-normal
	Burst bytes	Pareto

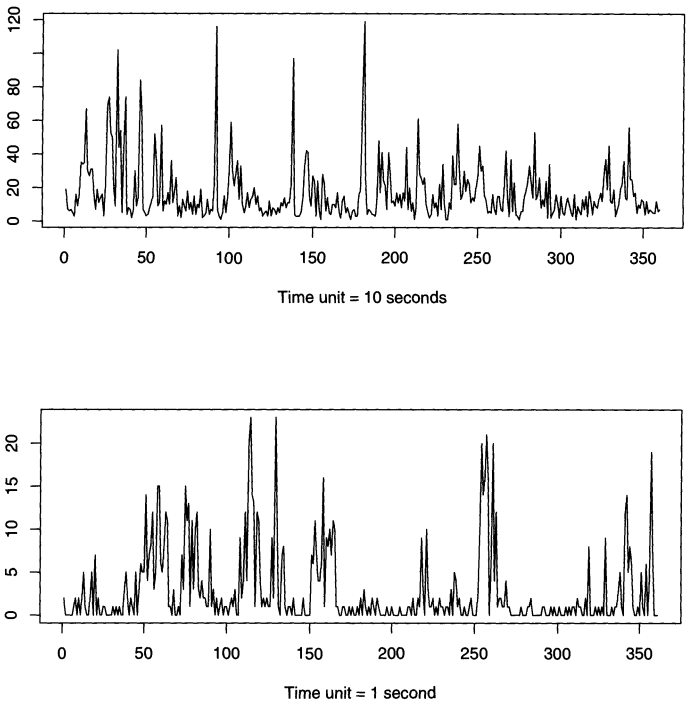

Fig. 2.7 Two views of the data from Figure 2.3. The number of packets per 10 seconds is depicted in the top graph, while the number of packets per second is depicted on the bottom.

This shows some evidence of self-similarity at two scales. Leland et al. [1994] show similar plots across five scales, ranging from a time unit of 100 seconds down to one of 1/100 of a second. This is very subjective and to strengthen the argument Leland et al. [1994] show similar plots for a synthetic data set based on a compound Poisson model fit to the data, for which self-similarity is not evident.

The concept of self-similarity can be made precise. Recall that the autocorrelation function of a process $X = X_1, X_2, \ldots$ is defined to be

$$r(k) = \text{cov}(X_t X_{t+k}). \tag{2.22}$$

Assume that X has a finite variance σ^2 and that $r(k) \sim k^{-\beta} L(t)$ as $k \to \infty$ and L is asymptotically constant. Processes with these properties are called self-similar. Define the aggregate time series $X^{(m)}$ to be the average of X taken over nonoverlapping blocks of size m: $X_k^{(m)} = 1/m(X_{km-m+1} + \cdots + X_{km})$. Denote the autocorrelation function of this process by $r^{(m)}(k)$. We say that X is second-order self-similar if $r^{(m)}(k) = r(k)$. Similarly, we call the process asymptotically second-order self-similar if the autocorrelation function of the aggregate time series converges to $r(k)$ as m goes to infinity. The parameter β gives a measure of the self-similarity, usually defined to be the Hurst exponent $H = 1 - \beta/2$.

Several papers describe the self-similar nature of network traffic. The interested reader is encouraged to investigate Feldmann et al. [1998], which analyzes data collected over several years, Crovella and Bestavros [1997], which looks at self-similarity for Web traffic, and Willinger et al. [1997] and Yang et al. [1999], who describe and apply the so-called "On/Off" model and variants to network data.

2.4 MAPPING THE INTERNET

Everyone knows the Internet is big. But just how big is it? How is it connected? If it had been designed, one could just go to the designers and ask to see the plans. However, the Internet was not so much designed as grown, and it is still growing. Floyd and Pacson [1999] give statistics on the growth, showing exponential growth through the 1990s. They are measuring traffic intensities rather than number of hosts, but the number of hosts shows similar growth.

The Internet Mapping Project

http://www.cs.bell-labs.com/who/ches/map/

is trying to map the Internet by running daily traceroutes. This allows them to construct a dynamic database of (nearly) all the machines on the Internet and the routes between them. These traceroutes are run from a number of different machines, in effect probing the Internet from different directions.

One of the results of this work is a set of very interesting and beautiful pictures of the Internet. Various color schemes add information to the graphs, such as coloring by domain or by latency. The database is available for researchers and is potentially of great use for those interested in studying the growth and extent of the Internet.

Displaying graphs of this size is not a trivial problem. These maps are laid out by a spring-embedding algorithm that basically simulates placing springs along the edges of nodes and solving for a minimal energy state. A good place to start learning about graph drawing is the book by Di Battista et al. [1999]. A paper discussing a particular technique for displaying large graphs is Wills [1999].

An interesting result of the network mapping project is shown in Figure 2.8. Since the Internet consists of machines all over the world, it is sometimes possible to link traffic on the network to events in the real world. The figure depicts the network maps from May 1 through May 6, 1999, for a part of the Internet that is found in Yugoslavia. As can be seen, there was considerable disruption on May 3. Some of the machines gradually came on line as the days progressed, but there was still some loss on the 6th. This period corresponds to a bombing campaign by NATO forces in neighboring Bosnia, which may have caused disruptions in Yugoslavia's power grid.

One possible modification to the display in Figure 2.8 would be to fix the position in the plot of each host throughout the graphs, perhaps with a rule to deal with new nodes that come on line, rather than recomputing the graph each time. Still, the effect is quite noticeable, even with the slight variation in node placement

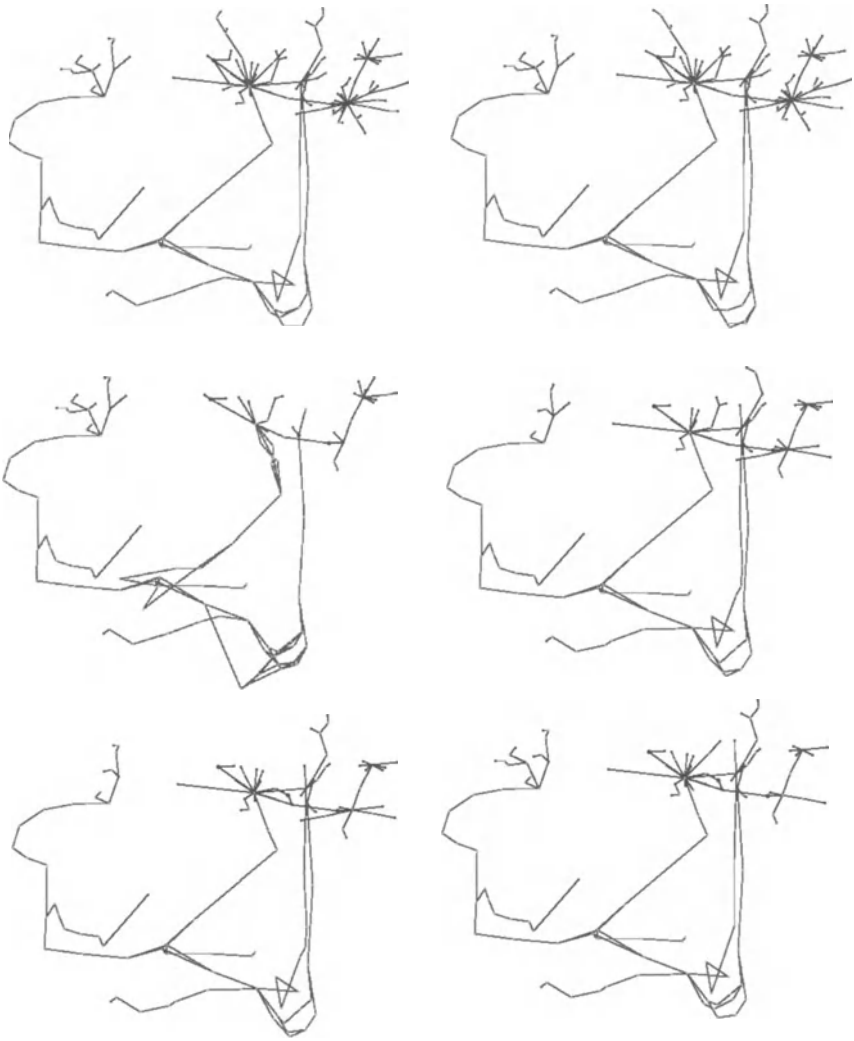

Fig. 2.8 A map of networks in Yugoslavia, showing the effect of the war on the networks. May 1 is in the upper left, with May 2–6 listed in lexicographical order.

across days. These images and a short movie, can be found at

http://www.cs.bell-labs.com/who/ches/map/yu/index.html.

The Internet weather report

http://www.mids.org/weather/

provides a view of the latency (round trip time) on the Internet as well as various statistics and other information gathered by scanning the Internet.

Another interesting site is

http://www.cybergeography.org/,

where whois lookups have been run to extract the mailing addresses of the owners of domains. These are then mapped onto a geographic database to produce a map of where the domains reside. Of course, this does not necessarily mean that the machines reside at those locations, but it gives an estimate of the spatial distribution of machines on the Internet.

Quite a bit of the information in the preceding references is available in the book by Dodge and Kitchin [2001]. This book considers many of the issues related to the mapping of the Internet, including sociological, geographic, and visualization issues. It also has extensive references and is a good starting place to learn about the various lines of research under way (at the time of its publication) in this area.

2.5 VISUALIZING NETWORK TRAFFIC

We have seen a number of ways to visualize network traffic in the preceding sections. In this section, we will look at some of these techniques in a little more detail and discuss some techniques that might not be familiar to many people working in computer security.

We will start with the simplest technique, the scatter plot. This will lead us to pairs plots, which are a way to display higher-dimensional data, which will in turn lead us to parallel coordinates. These techniques will be illustrated on network data.

2.5.1 Scatter Plots

The simplest (and arguably most powerful) technique for visualization of data is the scatter plot. In this technique, bivariate data are plotted as points in a plane, with coordinates corresponding to their values. We have already seen this in previous sections (for example, Figure 2.3). Another example, Figure 2.9, depicts about a minute and a half of incoming TCP packets to a site. Time is depicted on the x-axis and destination port is on the y-axis. Note that sessions are quite

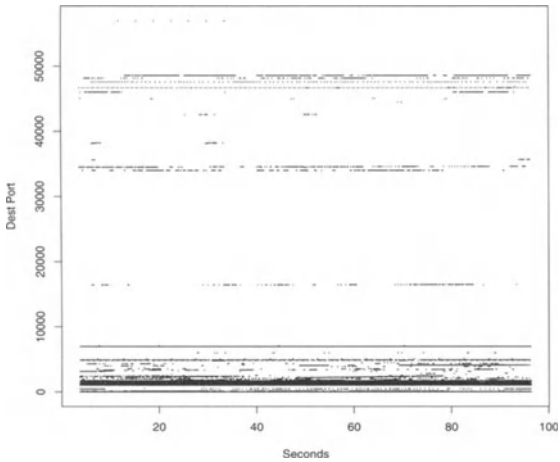

Fig. 2.9 Scatter plot of 93 seconds worth of data incoming to a site. In this case, there are 67,134 observations. The destination port number is plotted against time.

clear in this depiction as horizontal line segments. We can also see banding effects corresponding to port ranges for popular applications.

It is instructive to zoom in on this plot. Figure 2.10 depicts the data for which the destination port is less than 10,000, which is where the bulk of the data lie in Figure 2.9. Here we see an interesting phenomenon in the range between 40 and 80

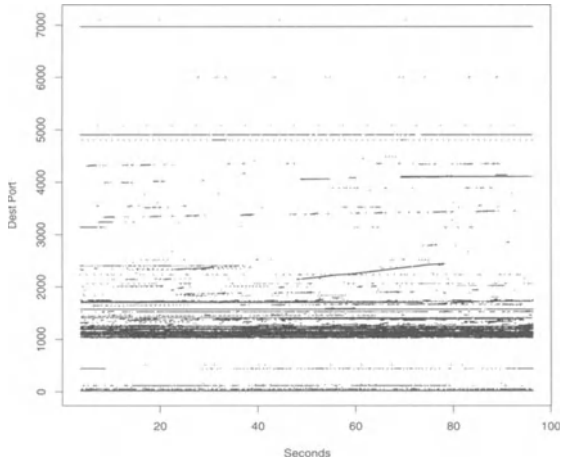

Fig. 2.10 Scatter plot of packets from Figure 2.9 with destination port number less than 10,000.

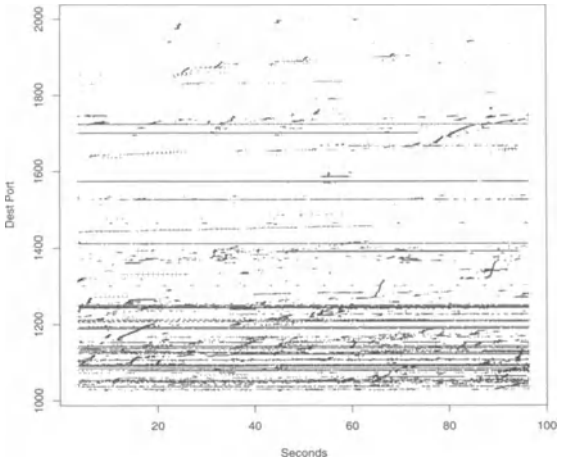

Fig. 2.11 Scatter plot of packets from Figure 2.9 with destination port between 1000 and 2000.

seconds. There is a line that is angled slightly, corresponding to destination ports between 2000 and 2500. Some further investigation shows this to be a Web session (the source port is 80). In this case, one of the site's users has gone to a Web server, and the downloads from the Web site are happening on the higher-numbered ports (which is typical Web client behavior).

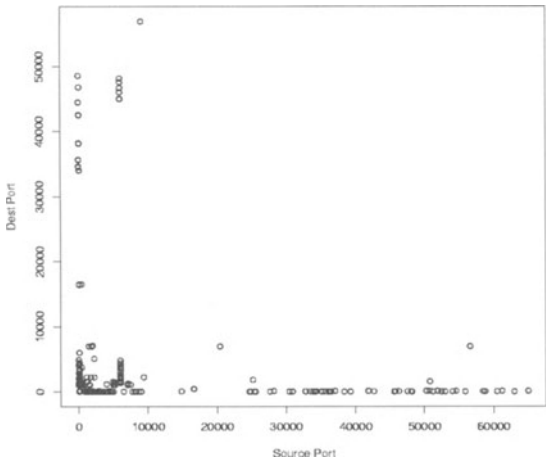

Fig. 2.12 Scatter plot of source port against destination port for the data from Figure 2.9.

This plot does not allow us to see that the Web session described above is actually a session to a single machine. In other words, although the line in the figure appears to indicate a single session we cannot tell that these packets are in fact related. They may be going to several different machines and the apparent correlation may simply be a coincidence. This is unlikely, especially given the length of the line but we cannot rule it out from this plot.

One way to solve this problem would be to use color to encode the destination (or source) machine. This would allow us to pick out sessions much more easily, but there are limitations to the number of colors that humans can reliably distinguish. Also, there can be problems with overplotting, especially when attempting to depict large time intervals for networks with heavy traffic loads. This requires interactive exploration where the user chooses a number of different views of the data; for example, zooming in to regions of interest.

A further zoom of these data is depicted in Figure 2.11. Here we can see quite a bit of fine structure. There are several angled lines, indicating data transfers of some kind, as well as a lot of horizontal lines, indicating application sessions on a number of ports. Again, color could be used to enhance this picture. Further zooms can also be used to explore different regions in the data.

Another way to look at these data is to plot source port against destination port. This is done in Figure 2.12. The "L" shape of this plot is indicative of the tendency for applications to use low-order ports. Thus, one can usually infer the application that corresponds to the packets as the one corresponding to the smaller of the two ports.

This is not a particularly useful plot. However, there are some outliers in the plot and by definition outliers are interesting. These outliers stand out quite easily in this plot, while they would be very hard to detect by looking in the raw data. A good rule of thumb is to look at any data via several different kinds of plots. Look for "interesting" structure and for "abnormal" or unusual data. The definition of unusual is subjective, but like art, one usually can recognize unusual data when one sees them.

Scatter plots are arguably one of the most powerful ways to visualize data, primarily because of their simplicity of interpretations. Their main drawback is the inherent low-dimensionality of the data that can easily be displayed in a scatter plot. In the rest of this section we will look at ways to plot higher-dimensional information.

Another drawback to scatter plots, which is common to all plots, is the problem of overplotting. There are only so many pixels on the screen, or dots on the paper, and so there are only so many distinct dots that can be represented. This problem can be partially addressed by binning the data and depicting the bins as is done with histograms, or by interactive displays which allow the user to zoom in to areas of high density. Unfortunately, paper generally does not lend itself well to user interaction.

2.5.2 Pairs Plots

An obvious extension of a scatter plot to higher-dimensional data is to plot each pair of variates in a separate scatter plot. This is the idea behind a pairs plot. We

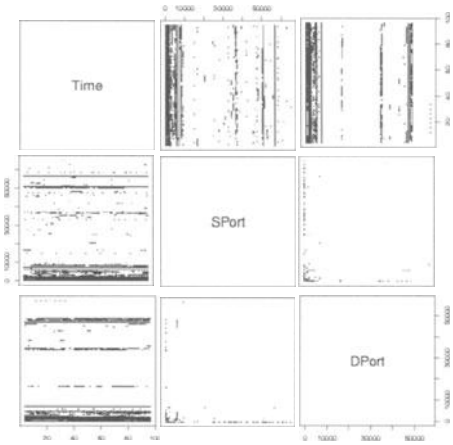

Fig. 2.13 Pairs plot 90 seconds of packets to a site. SPort and DPort represent the source and destination ports, respectively.

reconsider the data discussed in Figures 2.9–2.12, plotted as a pairs plot in Figure 2.13. The pairs plot is symmetric about the diagonal, so we are actually plotting twice as many plots as we need. However, sometimes it helps to see things from two perspectives, so I will use the default plot of the R pairs function in these plots.

We can see from these plots that the low-value destination ports span both time and source port, whereas the high destination ports are only associated with relatively low source ports. This corresponds to our earlier analysis. What cannot

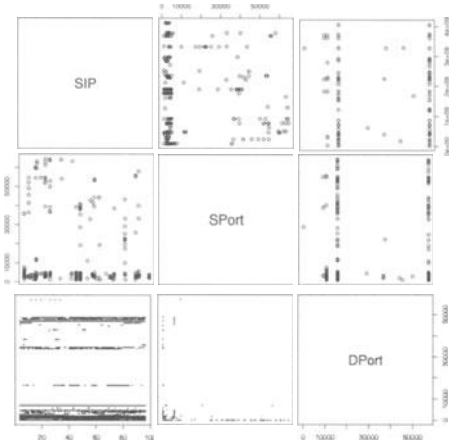

Fig. 2.14 Pairs plot of email (port 25) connections. SIP and DIP represent the source and destination IP addresses, while SPort is the source port. In this case, all the destination ports are port 25.

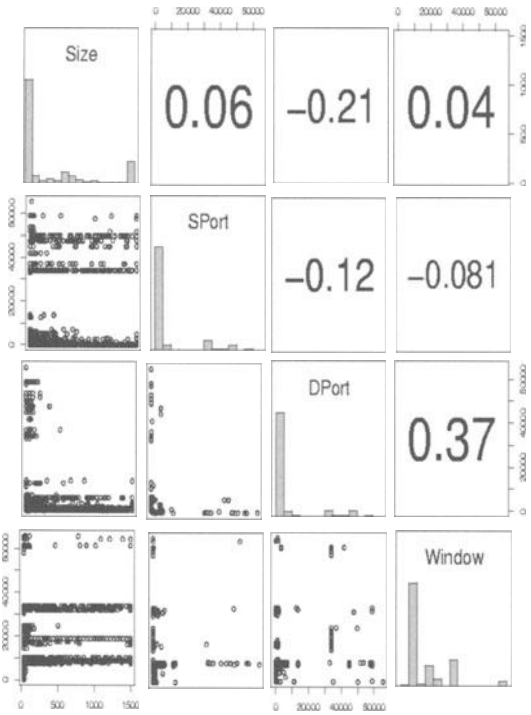

Fig. 2.15 Pairs plot depicting the packet size, source and destination ports and window size for 13,680 TCP packets. A histogram of each variate is plotted down the diagonal, and the correlations are given in the upper triangle.

be determined with single scatter plots is the combined information of all three variables. For example, consider the largest destination port in the SPort by DPort plot. Looking at this value in the Dport by time plot, we see a very regular pattern (points spaced roughly 3.5 seconds apart). This turns out to be activity between source port 9100 and destination port 56,946. This is an interesting pattern. Further investigation of the data determined that the session was initiated by the monitored site, and the temporal pattern was probably a result of load at one end or a delay in the route (other data between these two machines did not show the 3.5 second delay between packets).

Figure 2.14 depicts data to port 25 (email) for one hour's worth of data. The three variables source IP, source port, and destination IP are plotted (source and destination IP having been converted to 32-bit numbers). For example, the plot in the upper right corner has vertical axis source IP and horizontal axis destination IP. From this, we can count roughly nine destination IPS (the resolution of the plot makes this an inexact count). We can see that most sources send to one of two destinations (presumably the main mail servers for the site). From the SIP by SPort plot, we can see that most machines use source ports below 10,000, although some use higher ports.

Personally, I am not particularly impressed with pairs plots, particularly for more than three variables. I find that I have trouble visually processing more than three plots at a time, and so tend to not use pairs plots for much of my data analysis. Occasionally, though, they do provide useful information and so are a part of my visualization toolbox.

The pairs plots depicted here have a lot of redundancy and unused space. This extra plotting area could be used much more efficiently. One use of the extra plotting area would be to utilize color. One could color the points according to a separate scheme in the upper triangle of the pairs plot than in the lower. Another use would be to provide a different zoom level. The R function "pairs" allows separate functions of the data to be plotted in the different panels.

One interesting idea is to put a histogram of each variate down the diagonal of the plot. This is illustrated in Figure 2.15. In this plot the histogram for each variable is displayed in the diagonal. Also, in the upper triangle are the correlations for the pairs. Thus, we can see that the window size is most correlated with the destination port while packet size is inversely correlated with destination port. These correlations are fairly low. Note that source port is uncorrelated with window size, as is packet size. This illustrates the fact that the window size is a feature of the application, and has nothing to do with the size of the packet.

Pairs plots can usefully display up to about a dozen variables, depending on the amount and complexity of the data to be displayed. Other methods are required for higher-dimensional displays. We will look at two such methods in the next sections.

2.5.3 Parallel Coordinates

As we have seen, it is difficult to visualize high-dimensional data. Since the variates are generally considered to be independent, our usual approach is to place the axes perpendicular to each other. This only works for two-dimensional data, however. As we saw with pairs plots, we can get some information by looking at several bivariate plots, but it is difficult to extract multivariate information about the data from these plots.

The idea of parallel coordinates (Inselberg [1984], Wegman [1990]) is to place the coordinate axes parallel to each other rather than perpendicular. This allows us to plot points as connected line segments between the axes. Figure 2.16 illustrates this. The coordinate axes in this plot are displayed as vertical lines. Each observation is plotted as a piecewise linear curve. For example, the observation $x = (1.1, 2.3, 1.3, 2.8, ...)$ would correspond to the curve that first connects the point 1.1 on the first axis to the point 2.3 on the second, then from there to 1.3 on the third, 2.8 on the fourth, and so on.

Figure 2.17 depicts the email connections to a site during one hour. The first and third axes correspond to the source and destination IP addresses as 32-bit numbers. The second and fourth axes correspond to the source and destination ports. The values have been scaled between 0 and 1 for plotting. 1251 4-dimensional observations are represented in this plot, corresponding to 1251 email sessions.

One can learn quite a bit from this plot. There are 11 email servers depicted, two or three main ones representing the bulk of the data. The difference between

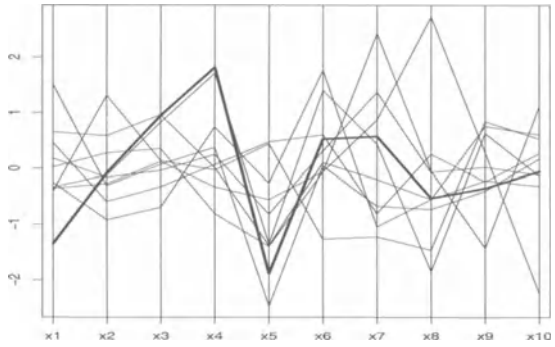

Fig. 2.16 Parallel coordinates plot of 10 observations drawn from a 10-dimensional standard normal density. One of the observations, corresponding to the point $x = (-1.35, -0.08, 0.95, 1.82, -1.88, 0.53, 0.58, -0.53, -0.36, -0.05)$, is highlighted in bold.

the number of email servers arrived at with this plot and that arrived at from the pairs plot (Figure 2.14) is a result of the low resolution of the pairs plot.

Machines sending email mostly tend to use relatively low source ports, although there are a number of machines that prefer higher ranges. The resolution is not

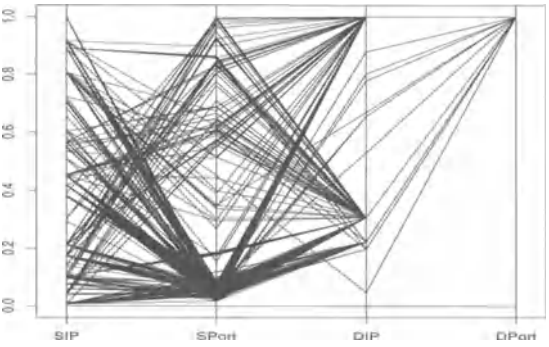

Fig. 2.17 Parallel coordinates plot of email (port 25) connections. SIP and DIP represent the source and destination IP addresses, while SPort and DPort are the source and destination ports. In this case, all the destination ports are port 25. The line at all zeros is not an observation but rather serves to delineate the minimum of the graph.

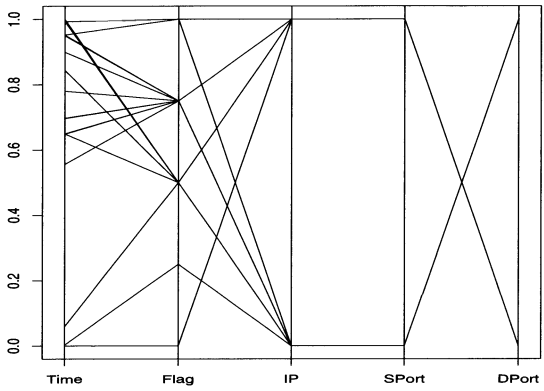

Fig. 2.18 Parallel coordinates plot of an email (port 25) session. The flags are, in increasing order on the axis, SYN, SYN ACK, ACK, ACK PUSH, ACK FIN.

good enough to determine whether the same machine will use low and high source ports (there are roughly 160 distinct machines represented by the SIP axis).

Figure 2.18 shows a single email session. In this plot, we have time as the first axis, followed by the flag combination set, the source IP, and source and destination ports (each either 25 or 1584). Since there are only two IPs in a session, we coded the mail server as 0 and the other IP as 1. By traveling up the time axis, we can clearly see the three-way handshake, followed by PUSHs and ACKs.

Parallel coordinates can be used to view data up to about 20–30 dimensions. They also suffer from problems of overplotting and interpretation. Ed Wegman has suggested using saturation brushing to deal with the overplotting problem and has provided some insights into interpretation.

Parallel coordinates plots, like all the techniques discussed in this section, can be enhanced by the proper use of color in the plots. This is an important aspect of visualization, that is often ignored in the literature due to printing limitations. This is changing, and as the cost of color reproductions drop, we will see a much more common use of color to enhance graphic displays.

2.5.4 Color Histograms

Another technique for viewing high-dimensional data is the color histogram, also called the data image. The idea is to treat the data as an image with, for example, the columns of the image corresponding to observations and the rows to variates. A simple example is given in Figure 2.19. Here, we have five measurements on each packet: the time of arrival (at the sensor); a binary value indicating whether the packet originated from the protected network; the source port; a binary value indicating whether the destination is the protected network; and the destination port.

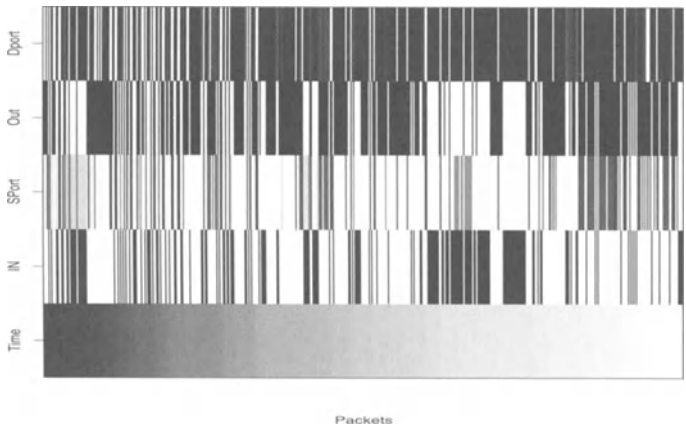

Fig. 2.19 A color histogram of SYN packets into and out of a site. There are 500 packets represented in this figure. The columns correspond to packets. The rows correspond to the variates, which are the time of arrival of the packet, whether the source IP is internal to the protected network, the source port, whether the destination IP is internal to the protected network, and the destination port.

Destination ports are scaled between 0 (black) and 1 (white), with values above 500 set to 1. Source ports are also scaled between 0 and 1, with 0 corresponding to a source port of 533 (the minimum in this data set) and ports above 2000 set to 1. Since it is difficult to separate the incoming and outgoing data in this graphic, we present these as separate images in Figure 2.20. We see that the outgoing sessions tend to be much more homogeneous than the incoming sessions, probably the result of a smaller pool of users, who are constrained by usage policies.

We will see more examples of color histograms and an extension, the data image, in Section 4.5.2.2.

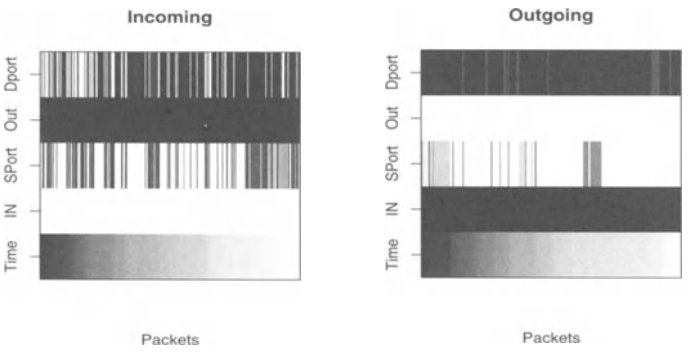

Fig. 2.20 The same data as in Figure 2.19 with the incoming and outgoing packets split into separate plots.

2.6 FURTHER READING

The problem of network tomography discussed in Section 2.2 can be approached from a number of other perspectives. A Bayesian approach is discussed in Tebaldi and West [1998]. This is a discussion article with two discussants, Vardi and McCulloch. Dinwoodie [2000], presents a Monte Carlo technique for computing the maximum likelihood estimates for the λ parameters of the Poisson distributions. In related work, Cao et al. [2000] describe a method using sliding windows to attack the network tomography problem.

Much work has been done on modeling network traffic, and we have just touched the surface with the discussion here. For example, Khalil et al. [1990] discuss the non-Poisson character of LAN traffic. Morris and Lin [2000] describe their work showing that although Web traffic is definitely not Poisson, aggregating it causes the behavior to settle down somewhat so that, while still not Poisson, the aggregated data scales in roughly the same manner as Poisson data. Many other papers have been written on these and related topics as a literature search will attest - far too many to list here.

To learn more about self-similarity measures and multifractal processes, investigate Riedi [1995], which provides some mathematical formalism. Another paper that looks at the fractal nature of network traffic is Addie et al. [1995]. Abry and Veitch [1998] use wavelets to analyze the multiscale nature of network traffic. Feldmann et al. [1997] also discuss self-similarity at the large and small scales for wide area network traffic. Fiorini [1999] looks at modeling heavy-tailed network traffic, from the perspective of analyzing its impact on quality of service. A similar issue is considered in Feldmann et al. [1999]. Gilbert et al. [1998] propose a method for visualizing multifractal scaling behavior. Roughan et al. [2000] provide a fast method for estimating the Hurst parameter and apply it to traffic modeling.

A slightly different perspective on network mapping is discussed in Theilmann and Rothermel [2000]. They provide dynamic distance maps, where the data collection is coordinated via hierarchical clustering of the hosts, to reduce the impact of the data collection on the network.

Much work has been done on modeling telecommunications traffic, and some of this is relevant to network traffic. Martine [1994] provides extensive discussion of these issues. A special issue of the Journal of Heuristics (Doverspike and Saniee [2000]) also has some articles of interest for understanding networks, although these focus more on the issue of designing rather than analyzing. Similarly, much work has been done on (vehicular) traffic analysis. Some of these techniques can transfer over, particularly those that are based on basic mathematics and computer science. For instance, see Foulds [1992], pp. 344–358, or Ettema and Timmermans [1997].

There are a number of good books on data visualization. Bertin [1967] is the classic, which appears to contain, in one form or another, every visualization technique ever invented (this is a slight exaggeration, but only slight). I highly recommend that anyone interested in the visual display of data take some time to look through this book. It is amazing the number of techniques depicted.

The classic series by Tufte (Tufte [1983, 1990, 1997]) is a very good place to get insight into the proper display of information. He discusses good and bad methods for representing data. He has many examples of displays designed so that the desired information is easily discerned, without distorting the true relationships between the data. These books are full of examples from a wide variety of problem domains and data types. The first book describes basic statistical graphics. In Tufte's nomenclature, it describes pictures of numbers. The second book is pictures of nouns. This refers to descriptions of evidence and data, particularly complex information. The third book contains pictures of verbs. This refers to illustrating cause and effect. These three books provide considerable information and advice for accurately and informatively depicting information.

The book Wainer [1997] is similar to the Tufte books and provides a good discussion of the issues of the informative display of quantitative information. It is quite a nice book, with all the visual impact of the Tufte books. I recommend it.

Another good book is Wilkinson [1999]. He develops a formal system for the graphical display of information, with many examples. In analogy with language, he describes a "grammar" of graphics, that ties mathematical and aesthetic rules together into a single framework.

Spence [2001] is another nice book on visualization. It has a chapter on the visualization of graphs, which is obviously relevant to the study of networks. There are a lot of nice color pictures generated by a variety of tools.

A very good paper on some of the mathematical issues in data visualization is Wegman et al. [1993]; see also Wegman and Carr [1993].

A very interesting phenomenon that is relevant to network modeling and analysis is the so-called "small world," or "Kevin Bacon" phenomenon. This is described in some detail in the book by Watts [1999]. The idea is best illustrated by the "Kevin Bacon Game": select any actor. If he or she was in a movie with Kevin Bacon, they receive a score of 1. If they were in a movie with another actor who was in a movie with Kevin Bacon, they score a 2, etc. The claim is that all (or very nearly all) actors have a score of not more than 7. Thus, the world of actors is a "small world." This is the "small world" phenomenon that is familiar to us all when we meet someone new (possibly on a trip far from home) and discover that we have a friend in common with them. In the Internet, this is relevant from the standpoint of determining, for example, how many links one must follow from one Web site to any other (is the World Wide Web a "small world"?). The work detailed in the book uses quite a lot of machinery from graph theory, and so is rather technical.

3

Evaluation

3.1 INTRODUCTION

Statistics involves the fitting of models to data and making inferences from these models. One is often interested in the models themselves because of what they may tell us about the underlying physical process that generated the data. Thus, much of statistics concerns itself with goodness of fit tests, confidence regions, and other tools for determining whether one's model appropriately and accurately describes the data, and for making inferences from the estimated model.

In pattern recognition the inferences that one wants to make are ones of assignment. For example, given a trace of network data, one wants to classify it as an attack, or not. Thus, while the model that one chooses is not uninteresting, the ultimate goal is a pragmatic one: how well does our model detect or classify attacks? As a result, one often finds that the tests of model fitness are reduced to tests of classifier performance. In this section we will look at some of the issues that are of interest in evaluating the performance of intrusion detection systems.

Any evaluation of a pattern recognition system requires the estimation of two quantities: the probability of detection (PD) and the probability of flase alarm (PFA). These are intertwined, and in general it is not possible to simultaneously achieve a PD of 1 and a PFA of 0. The idea is that there is a "target" class that one is interested in detection. For example, an intrusion. The probability of detection is the probability of correctly detecting the presence of the target class. A false alarm occurs when a detection is declared even though the target is not present. For example, declaring an intrusion has occurred when none did.

In statistical terms, the probability of a false alarm is related to the type I error (rejecting the hypothesis when it is true). The null hypothesis in this case is that no attack is present. Thus, a type I error would occur if we incorrectly labeled a

benign event as an attack. The PFA is the probability of making a type I error. The probability of detection corresponds indirectly to the type II error (failing to reject the null hypothesis when it is false). In our case, this means failing to detect an attack when one is in fact present. Thus the probability of detection is 1 minus the probability of making a type II error.

Some authors use the terms recall and precision rather than probability of detection or false alarm. "Recall" corresponds to the probability of detection. This is the probability of correctly "recalling" the target class. Precision is one minus the probability of false alarm, or the probability of correctly stating that a target is not present.

Let us consider the two class problem. Assume we have designed a classifier that discriminates between two different classes and that we have identified one of the classes as "target" (or, in our setting, "intrusion"). We wish to determine how well it performs. Assume further that we have an independent data set with which to test the classifier. We evaluate the classifier on the data and determine the proportion of target observations correctly classified (PD) and the proportion of nontarget observations incorrectly classified as target (PFA).

For most classifiers there are parameters to adjust (for example a detection threshold) that can affect the performance of the classifier. Varying these parameters produces slightly different classifiers with slightly different PD and PFA values. By varying these parameters and obtaining a range of values, we can plot a curve of PFA vs. PD, the so-called Receiver Operating Characteristic curve, or ROC curve.

One generally tries to choose the classifier with the best PD within constraints on the acceptable PFA. This is chosen based on the relative costs of the two types of errors: missing an attack versus analyzing a false alarm. Given an ROC curve, one can select the parameters that produce the desired classifier.

This becomes a bit more complicated when one considers problems with more than two classes. For example, one might want to classify the type of attack into one of several groups, rather than simply announcing the detection of an attack. This can be formulated as a sequence of two class problems, either as "class i" against all others, or pairwise, "class i" against "class j" for each pair i and j. Consult a book on pattern recognition for more discussion of these approaches.

There is a school of thought in the computer security domain that says there is no such thing as a false alarm. The assumption here is that in a well-designed system, any alarm contains information. Different alarms require different levels of intervention. For example, one may see a few packets that look like a probe for vulnerable systems. The security officer may want to know about this, even though it is not yet a problem and even though in reality it may not be a prelude to an attack at all. Proctor [2001], pages 108–111, discusses this in some detail. It is worth keeping this in mind when evaluating an intrusion detection system.

Evaluation of intrusion detection algorithms is problematical for several reasons. First, it is difficult to collect data representative of the threat. Since the threat is constantly changing as new attacks (and vulnerabilities) are developed, it is vital that an IDS be able to detect novel attacks. It is well known to be difficult (and perilous) to make predictions outside one's data, and this is precisely what is

expected of IDS evaluations. Worse, the data are by definition nonstationary, so any evaluations are of transient utility.

Second, if one collects real data, one can never be sure that there are no subtle attacks hiding undiscovered in the data. This affects both the calculation of the probability of detection and the probability of false alarms. You cannot count missed detections that you do not know about, and you cannot be absolutely sure that a false alarm is not in fact a correct detection. Further, in order to get a good sample of attacks, you may have to collect a very large amount of data. One way around this is to embed real attacks in the data (either artificially or by attacking your own network).

Finally, few intrusion detection systems are truly automated. For example, the SHADOW system (Section 4.4) fundamentally relies on an analyst to process the suspicious events and generate the reports. One could simply count each suspicious packet (or block of packets for a given source host) as a candidate detection and compute false alarms and detections from these. However, this would overestimate the false alarm rate. Alternatively, one could treat the analyst as part of the overall system and evaluate the performance of the system with the human in the loop. Of course, this adds another level of variability (analyst expertise) that must be controlled for.

Still, it is essential to do careful evaluations. None of these problems are unique to intrusion detection, they are simply more obvious than in some other domains. We will look at two methodologies for evaluating intrusion detection systems, but first we must discuss evaluation of classification systems in general.

3.2 EVALUATING CLASSIFIERS

We have discussed one method of evaluating a classifier, which is to use an independent test set. Thus, one collects data and then separates the data into a "training set" and a "testing set." The classifier is then designed using the training set. Recall that a classifier is a rule that assigns a class label to observations. In order to devise the rule, example observations for which the class labels are known must be available. These observations are called "training points." Using training points, the parameters of the classifier can be adjusted to maximize the performance of the classifier (on the training points). Once the classifier has been constructed, its performance on the test set is measured, providing an estimate of the performance of the classifier.

Care must be taken to ensure that the sets are indeed independent and that an unconscious (or conscious) bias is not introduced. For example, one may (accidentally or on purpose) place all the "difficult" points in the training set, thereby biasing the estimate toward better performance. One way to avoid this is to randomly split the data to reduce selection bias. Then, after using the two sets to evaluate the classifier, reverse them (the training set now becomes the test set and vice versa) and redo the evaluation. An extension of this idea is to repeat the random selection of training/testing observations many times to get an average performance measure that avoids the problem of selection bias.

A related idea is that of cross validation. In its extreme form (1-point cross validation), the test set consists of a single observation. The classifier is built on all the remaining observations and then its performance on the single test observation is measured (this is a binary response: it either gets the observation's class right or it does not). This is then repeated with a different observation until all the available observations have been used as test observations.

This can be generalized to k-point cross validation. The data are split into subsets of size k, and each subset in turn is used as a test set, while the rest go into the training set. The other extreme from the 1-point cross validation is the $n/2$-point cross validation, where the data are split evenly into two sets. This is the training/testing described earlier.

Another terminology is sometimes used in cross validation. Instead of "leave k-out," one will sometimes see the phrase "k-fold" cross validation. The difference is that in k-fold cross validation one splits the data into k subsets of equal (or nearly equal) size. Then one subset is withheld, the classifier is trained on the remaining subsets, and the withheld set is used as a test set. This is repeated, in the same manner as described above. Thus, k-fold cross validation is essentially n/k-point cross validation. This can be slightly confusing the first time one comes across it.

The collection of the data to be used for training and evaluating the classifier is a nontrivial task, particularly for intrusion detection. Imagine setting up a sensor on a network, collecting data for (say) a month, then using the data for the evaluation. A few obvious questions come to mind:

- What is "truth"? In other words, which packets or sessions are intrusions and which are not?
- Is this a typical data set? One collected at a university in July might not be representative of the network in November.
- How long are the data going to be representative (if they are)? How fast is the network (and the threat) evolving?
- Have we measured the threat? Are the attacks representative of the ones we want to detect?
- Along the same lines, are the ones we currently know how to detect really all of them?

There is a (possibly apocryphal) example of this from an image processing problem. A classifier was built to distinguish images that had tanks in them from those that did not. The classifier worked quite well. However, when an independent set of images was subsequently produced, the classifier was no better than chance. Upon investigation, it turned out that the images in the original data set that contained tanks were all taken in the morning, whereas those without tanks were taken around noon. The classifier was detecting the brightness of the images. Low-light images had tanks in them (in the training set) so all one needed to do to detect tanks was take a light level!

Another example involves a radar that was used to detect tanks. In order to identify the tank it was necessary to determine its orientation. To collect data from

a variety of orientations the tank was mounted on a large turntable and data was collected from a variety of orientations relative to the radar. It subsequently turned out that the best discriminator of tank orientation was a return that was generated by a corner of the turntable. This clearly is of little practical utility, unless one can convince the enemy to mount their tanks on turntables.

This sounds silly, but it is a real concern. Suppose you want to detect buffer overflow attacks against network applications such as telnet. If all your training examples are attacks against telnet, there is a very real possibility that the classifier will learn to detect something that is related to telnet (and possibly unrelated to buffer overflows).

There are some studies in the pattern recognition literature in which the same data are used to evaluate the classifier as were used to build it. This is called "resubstitution." It would seem that this is a very dumb idea from what has been said previously. However, it is a valid method of evaluation of classifiers. It is biased, but it does provide an estimate of performance. It is probably best to avoid it, though, because it will tend to give optimistic estimates of performance. For the mathematically inclined, Devroye et al. [1996] has a short chapter devoted to resubstitution.

An example of a classifier where resubstitution should never be used is the nearest-neighbor classifier. In this classifier, the observation is given the class associated with its nearest neighbor from the training data. However, if the test observation is in the training data (as it would be with resubstitution), then it will always get the observation right (assuming the observations are all distinct). Thus, the resubstitution estimate of performance for the nearest-neighbor classifier is perfect! The PD is 1 and the PFA is 0. Note that this is the case *regardless of the problem or the training data.* Obviously, this is not a particularly accurate estimate of performance for this classifier.

Note that even with k-nearest neighbor (where the classifier takes a vote amongst the k training observations closest to the point) the performance estimate will be severely biased by the resubstitution method. The effect would be as if one were to take a vote among k people, where one of them always knew the right answer. This will give a very optimistic estimate of the performance of the classifier. In either of these cases, however, it is easy to turn the resubstitution estimate into a leave 1-out cross validation estimate assuming the observations are distinct: instead of taking the k closest, take the $k+1$ closest, but drop the closest from the vote.

There is another problem with the detection and classification of rare events. Even if you have a very good detector, with a very small false alarm rate, it could be that a large proportion of the "detections" are in fact false alarms. This seemingly counter intuitive result can be derived as follows.

Recall Bayes' Theorem (Hogg and Craig [1995],):

$$P(C_j|C) = \frac{P(C_j)P(C|C_j)}{\sum_{i=1}^{k} P(C_i)P(C|C_i)}. \tag{3.1}$$

In our context, let C be the event that our classifier tells us it has detected an attack (raises an alarm), C_1 be the event that it really is an attack, and C_2 the event that

it is a false alarm (for this example, $k = 2$). Then, we have, from Equation (3.1), letting I indicate an attack (intrusion), A indicate an alarm, and $\neg$ indicating the logical negative ("not"):

$$P(I|A) = \frac{P(I)P(A|I)}{P(I)P(A|I) + P(\neg I)P(A|\neg I)}. \tag{3.2}$$

Now, let us look at some reasonable values. Suppose a network logs 1,000,000 packets per day, and of these 20 packets per day correspond to attacks (on average). That says $P(I) = 20/1,000,000 = 1/50,000$. Suppose that our detection rate is 99% and false alarm rate 0.1%. This says that $P(A|I) = 0.99$ and $P(A|\neg I) = 0.001$. Plugging these in, we have $P(I/A) = 0.019$, or about 2%, so only 1 alarm out of 50 is an attack. If we can get our false alarm rate down to 0.01%, things look a little better: $P(I/A) = 0.1653$, or about 17%. This is the "base-rate fallacy" described in Axelsson [1999] and Axelsson [2000]. It takes a false alarm rate of 0.001% (a probability of 10^{-5}) to bring our probability of intrusion given an alarm up to 66%. A security officer may very well ignore a system that is wrong 49 out of 50 times and might very well be disgusted with a system that is right only 2/3 of the time, but as we have seen, it takes an extraordinarily good system to obtain this level of performance.

This might seem strange until you realize what is actually happening here. Since we only care about alarms, we are ignoring the vast majority of packets. Thus, a system with a false alarm rate of 0.00001 and detection rate of 0.99 detects (essentially) all of the 20 attacks, and roughly 10 extra packets. Put this way, this seems quite reasonable (in fact, it's outstanding). The security officer mentioned earlier might consider 66% to be perfectly fine. With a false alarm rate of 0.001, this grows to 1000 extra packets. Now, our security officer may simply learn to ignore the system. The problem lies in the vast number of "normal" packets. This explains the focus of many research efforts on the reduction of false alarms.

The reduction of false alarms is particularly important in computer security. As we will see, there are intrusion detection systems designed to detect network intrusions (Chapter 4) and those designed for detection on a single host (Chapter 5). We will see that some attacks cannot be detected at the network level, whereas other attacks are best handled at this level. False alarms at these two different levels have dramatically different consequences.

For example, consider a network consisting of 100 machines. This network has a network monitor at the firewall looking for network intrusion attempts such as probes and mapping attempts (Section 4.3.2) or denial-of-service attacks (Section 4.3.1). On each host is a host-based intrusion detection system (IDS). Assume that the people using the systems know nothing about security, so the attacks must be reported to the site security officer (SSO). Now, consider the false alarm rates. If the network IDS has 20 false alarms per day, then the SSO has 20 alarms that must be tracked down and identified. If a host-based system has two false alarms per day, then the SSO has 200 alarms to track down. Clearly, host-based systems must either be handled by the individual owner of the system (an ideal that does not appear to be attainable for the majority of organizations) or must have a much lower false alarm rate (as measured by alarms per day) than network IDS systems.

3.3 ROC CURVES

Most intrusion detection systems are primarily focused on the problem of detecting attacks. Thus, they are two-class classifiers, with the two classes being "attack" (class 1) and "not an attack" (class 2). Some go further and try to determine what kind of attack it is and what the potential consequences are, but first and foremost is the detection of the attack.

As we have seen, the two numbers of interest are the PD and PFA. However, one generally cannot simply state the PD and PFA that one wants and design the algorithm to provide them. If one could, one would always require a PD of one and a PFA of zero. Consider for illustration the nearest neighbor-algorithm, with the following twist: take the distances to the nearest observations from each class, and consider the ratio. Thus,

$$L(x) = \frac{d(x, C_1)}{d(x, C_2)}, \tag{3.3}$$

where d is some distance metric and $d(x, C_i) = \min(d(x, c)|c \in C_i)$. Using Equation (3.3), we have the standard nearest-neighbor rule: Assign the point to class 1 if $L(x) < 1$; otherwise, assign it to class 2.

Putting the algorithm into the form of Equation (3.3), however, allows us to adjust the rule to change the PD/PFA. By considering $L(x) < \tau$ for various values of $\tau < 1$, we require the classifier to be "more sure" of its answer and thus (potentially) decrease the PFA (possibly at the expense of decreasing the PD). Similarly, considering values of $\tau > 1$ allows us to insist that the classifier call it class 1 as long as there is some chance that it is an attack, increasing the PD, while at the same time (potentially) increasing the PFA. By choosing different values for τ and computing the PD/PFA, we can produce a plot of PD vs. PFA. This is an ROC curve.

3.4 THE DARPA/MITLL ID TESTBED

DARPA, the Defense Advanced Research Projects Agency, had a program in computer security and intrusion detection and wished to obtain reliable estimates of the detection and false alarm rates of competing algorithms and systems. In order to do this, MIT Lincoln Labs (MITLL) was contacted to build a simulation network. This would simulate network traffic into which attacks could be injected. This eliminates the second problem mentioned earlier: there could be no "unknown" attacks in the data. MITLL, and hence DARPA, would know everything about all the attacks and other traffic.

In order to model network traffic, 4 months' worth of traffic was collected at an Air Force base and analyzed. From this, the percentage of email, Web, and other traffic was determined, as well as other information required for the model. The simulation model consisted of models for different types of users (secretaries, managers, etc.), so that a representative mix of the types of traffic would be obtained.

In order to model Web traffic, actual Web pages were downloaded that were representative of the kinds of accesses seen in the Air Force data. Web surfing sessions were then simulated throughout the network, with virtual machines acting as the Web servers.

Email was simulated by generating random messages with the statistics of English messages. It is not clear how the generation of these emails affects the false alarm rate for systems that search email text for included viruses or other attacks.

By producing a virtual network, MITLL was able to simulate a very large network on a small number of real machines. This was a very cost-effective scheme.

The MITLL approach is probably the best way to simulate network traffic. Floyd and Paxson [1999] argue that we do not have enough information about the underlying statistics of the Internet, and thus that modeling Internet traffic is inherently extremely difficult. They conclude that the best way to simulate network traffic is by focusing on "source-level" traffic rather than packet-level traffic. This is essentially what the MITLL simulation does by focusing on the applications and simulated users rather than constructing individual packets. I do not know whether the statistics of the packets (sizes, arrival times, etc.) is representative of the true distributions of Internet traffic, but it is not clear that this level of detail is relevant to intrusion detection systems. Cabrera et al. [2000] showed some ability to detect intrusions by looking at deviations from traffic intensities (telnet session arrival times) using a Kolmogorov-Smirnov test to test for deviations from a Poisson distribution. This does indicate that at least for this kind of statistic the simulated data seem to agree with Internet traffic statistics.

One issue that is relevant to the MITLL data is the question of how well the data simulate real networks. As an anecdote, several of people who install SHADOW sensors have told me that the first thing that must be done once a sensor is put in place is to track down all the misconfigured hosts and routers on the network. They always see a lot of broadcast packets and other indications of misconfigurations. At NSWC, we had an instance of a machine in Colorado that kept trying to mount a disk on one of our machines. This was the result not of an attack but rather of an error in the IP address. The experience is that real networks are noisy. This noise can have an impact on the false alarm rate of intrusion detection systems.

The first part of the MITLL study involved diseminating data to researchers involved in the evaluation. Several weeks of data were generated and given to researchers to tune their algorithms. For these data, all attacks were clearly marked. The researchers were also given any information about the protected network that they desired. Data were of several types. There was network (tcpdump) data, log data, and file data (such as file sizes, times and dates, and so on).

In the first evaluation, several algorithms were installed at MIT Lincoln Labs as if they resided on the virtual network. In later tests, researchers were given several weeks of test data in which no attacks were identified. The systems were then required to provide DARPA with their detections in an agreed-upon format. They were encouraged to provide confidence numbers rather than binary responses, but most algorithms in the early evaluations produced binary responses, so ROC curves could not be computed.

The first results of the DARPA evaluation are reported in Durst et al. [1999]. This is a preliminary report of the first attempt. The evaluation process is ongoing, and a number of problems with the first evaluation have been ironed out in subsequent evaluations. For example, there was some controversy about scoring some systems: do you count a detection if you flag some (but not all) of the attack as suspicious? If you do not correctly classify the attack, but do flag it as an attack? Differences of opinion on the scoring resulted in slight differences between the claims of researchers and the results that DARPA reported.

The results of this first evaluation are not encouraging. The best algorithms operated at a detection rate of about 25% with a false alarm rate of about 0.1%. A PFA of 0.1% on network traffic is unacceptable for most large networks, even though this number is computed on a per-session basis rather than on individual packets.

More extensive evaluations have been performed since this first attempt. These evaluations are reported in Lippmann et al. [1999]. Although the systems tested continue to improve, the results are still not encouraging. The systems have a difficult time with new attacks (not surprisingly), and they are not yet performing (as of this evaluation) at the level DARPA has set as the goalposts for IDS systems.

The most difficult task that the DARPA evaluators have set for themselves is to evaluate the performance of algorithms in the detection of novel attacks. In order to perform this evaluation, the DARPA evaluators developed several new attacks, which were not provided to the researchers in the training data. An obvious question is how representative these novel attacks are of real attacks.

Another issue of concern in evaluating intrusion detection systems is to ensure that systems are not penalized for missing attacks that they could not detect. For example, the email viruses (see, for example, Section 6.7.3.1) cannot be detected by a network monitoring system that only considers the packet headers. Thus, the evaluators must determine the class of attacks that a given system can be used to detect and only score a system on attacks appropriate to the system. To this end, researchers were required to inform DARPA of the data used by the system and the kinds of attacks that the system could be expected to detect.

There is a subtlety in the definition of "attack" that needs to be considered in evaluating intrusion detection systems. The problem is that in some cases it is intent that determines whether some activity constitutes an attack. The traceroute "attack" is a good example (see pages 109 and 130). In this, the attacker uses the traceroute utility to determine all the routes to a site. This can be used to determine potential bottlenecks or downstream sites that can be attacked to shut off the target site from the Internet. Traceroute can also be used to map out the routers within a site. Unfortunately, traceroute is a commonly used utility, and there is no way to tell the intent from the packets.

Some security analysts want to know when a host (or set of hosts) sends a large number of traceroutes to their site, whereas others don't want to be bothered. If a system reports traceroutes when they are not part of an attack, is this a false positive? Is it a missed detection if the system does not report such an event and it turns out to be an information gathering attempt prior to an attack? Since one cannot determine intent from the packet trace, there is no way to distinguish

between benign traceroutes and those intent on information gathering unless one can correlate the packets with others that are also gathering information.

The problem is particularly germane to the DARPA approach of using simulated data. One can easily set most systems to ignore specific events (like traceroute). This kind of tuning is always done to configure the system for a specific network environment and security policies. However, unless the security policies are very specific and the virtual network environment well known, some systems may produce poor results as a consequence of not being properly configured for the test environment. To their credit, DARPA and MITLL have considered these issues quite carefully in their evaluation.

The most recent published evaluation of the DARPA work is Lippmann et al. [1999]. Six different research groups participated. Each group was given a training data set in which attacks were identified and information about the protected network was provided.

The groups were subsequently given test data in which attacks were embedded but not identified to the researchers. Thus, the evaluation was blind. The researchers had to provide their system's detections, which were then used to evaluate their performance.

As described earlier (and in more detail in Lippmann et al. [1999]), a considerable amount of work went into designing the victim network. Users were simulated (and in some cases real users interacted with the network), different applications were run on the network (Web, FTP, telnet, email, etc.), and the traffic from these applications was sometimes generated according to models developed by DARPA and MITLL and sometimes real sessions were taken and inserted on the virtual network.

There was one potential flaw in the design of the experiment, however. There were three Unix machines that were the victims of the bulk of the attacks (this study involved only Unix machines): a Linux machine, a SunOS and a Solaris. (A router was also a victim for some of the attacks.) If these machines were also those attacked in the training data (Lippmann et al. [1999] are not clear on this point), then there is the potential that algorithms could learn this (recall the tank anecdotes from before) and ignore packets sent to other systems. Even if this were not the case, it appears that only these three systems provided audit and file data, so it would be reasonable even for those that did not use these data to assume that they were the only machines attacked.

Recall the tank anecdote. I do not mean to imply that any of the researchers used this information in designing their algorithms or in providing the results from their algorithms to DARPA. However, a sufficiently sophisticated algorithm might be able to infer that only a subset of the machines are ever attacked and incorporate this into its algorithm, unbeknownst to the researchers. Even if this did not happen in this experiment, it is something to keep in mind in future evaluations.

3.5 LIVE NETWORK TESTING

The DARPA approach has some very strong advantages over live data. For one, all the attacks are known by the evaluation team (although not by the systems being

evaluated), so true detection and false alarm values can be calculated (with the caveats mentioned in the previous section). The size of the network, volume of traffic, and type of traffic can be adjusted as desired. Any desirable data can be captured. The entire network is known down to the level of the operating systems and users on the systems.

One problem with any simulation is to determine how well it simulates the real world. Another is that even the best simulation is only modeling a specific network environment, which may not be typical. The results may not be relevant to other environments.

Another problem with a simulation is the modeling of novel attacks. Without some model of attacks, it is very difficult to model new ones. DARPA and MITLL have tried to create new attacks, and judging from the results of their studies, they have done a pretty good job from the perspective of creating attacks that are hard to detect. However, by their very nature, new attacks are often ones that nobody predicted.

A good example, which we will see in Section 6.7.3.1, is the Melissa virus. Unless one knows to look for certain types of macro-language commands within attachments to email, this virus is going to be hard to detect. More importantly, it requires quite a bit of imagination to come up with such an attack the first time. Now that Melissa has made the news, these kinds of attacks are quite common. However, how does one determine the probability that a system will detect next year's "Melissa" attack? In other words, how does one determine the probability of detecting the next new type of attack?

Live network testing tries to answer the questions related to real-world performance measures. By running real network data through a system, one can determine the answers to the following questions:

- Does the system have an acceptable alarm rate? This means that the number of alarms that turn out to be false is small enough, on the given network, that the SSO can handle them.

- Does the system detect the attacks that it should? In a well-designed study, one can have several different systems running, along with an SSO that understands the network and the typical threats, and alert system administrators that are on the lookout for attacks. Thus, one can detect the kinds of attacks that are known, and can determine whether the tested system is detecting them. This does not address the problem of novel attacks.

- What kinds of attacks are missed? With several systems looking at the network, plus a knowledgeable SSO and alert system administrators, most of the attacks on a given network or machine can be detected. This level of alertness may not be maintained over the long term, but for short periods of time, with sufficient effort, one can do a pretty good job of detecting all but the most subtle attacks.

- What kinds of attacks are detected? A "red team" can be used to attack the network, ensuring that real attacks are mounted. This can also allow for interaction between the red team and the researchers to try to determine the

blind spots in the systems being evaluated and suggest methods to improve them.

- Does the system provide sufficient information about a suspected attack that the true nature of the incident can be determined? Some systems may not collect or retain the data necessary to do this.

- Can the system handle real data rates and all the problems that occur on real networks?

Obviously, true false alarm and detection rates cannot be determined on a live network, but they can be estimated, and it is not clear that these estimates are worse than those obtained from simulations. As an analogy, consider the problem of constructing a system to detect breast cancer from digital mammograms. One needs a training set. This is a bit tricky. One could take all mammograms of women who have had their cancer confirmed by biopsy, but this assumes that the cancer was detected, and hence at best the training set consists of those cancers we already know how to detect (although perhaps not as reliably as we would wish). What images should we use as "clean" images? Again, if we call those for which no biopsy was done "clean" we have no idea how many missed detections we are allowing into our training set. It is infeasible (and unethical) to biopsy apparently healthy breasts simply to construct a good training set, so one must live with the fact that there may be missed cancers in both the training set and any test set we use to evaluate the classifier. Imagine how difficult it would be to get FDA approval for a system for the detection of breast cancer that had been designed and tested exclusively on simulated mammograms.

Similarly, in real life, unless the intrusion detection algorithm can be constructed from first principles (for example, using the RFCs to determine legal/illegal behavior), it must be trained on real data, and these real data may contain missed attacks. It seems reasonable that such systems should be trained and evaluated on real data to get the best estimate possible of how they really work. This is not to imply that simulation results are not extremely valuable, but it is my opinion that they cannot completely take the place of real-world evaluations on real networks.

3.6 FURTHER READING

There are several good books on pattern recognition, and most of them discuss classifier evaluation. One of the classics in statistical pattern recognition is Fukunaga [1990]. Another classic that has been expanded and reissued in a new edition is Duda et al. [2000] This is an excellent book that I highly recommend.

A book that discusses evaluation of classifiers at length is Hand [1997]. Devroye et al. [1996] has a good, but quite theoretical, treatment of pattern recognition theory.

Both Axelsson [1999] and McHugh [2000] give extensive critiques of the DARPA intrusion detection evaluations. At the time of this writing the McHugh paper is not yet published, so I am relying on descriptions of the paper. Axelsson

[1999] also discusses and critiques other evaluations that have appeared in the literature.

Part II

Intrusion Detection

4

Network Monitoring

4.1 INTRODUCTION

Network monitoring involves attempting to detect attacks on a network, or on hosts on the network, by monitoring the network traffic. This is usually done at the firewall or filtering router, so that all traffic coming into the network can be analyzed.

One of the best introductions to network monitoring can be found in Northcutt's book (Northcutt [1999]). This is one of the few books, as of the time this is written, that give explicit details on how one can detect intrusions at the network level.

We will start by describing one of the tools used for network monitoring, the tcpdump program. We have seen how to use this program to monitor network traffic in Section 1.2. In Section 4.2 we will look at how filters can be defined to specify the types of packets of interest. This will give us the ability to scan tcpdump files (or live network traffic) for packets that are indicative of certain attacks.

In Section 4.3 we will consider some specific network-based attacks. Each attack will be described in detail, with examples of network traffic illustrating the attack when applicable and examples of tcpdump filters designed to detect the attack when possible.

Although we will not detail any commercial network security products, we will look at a freeware program, SHADOW, which is the result of work at the Naval Surface Warfare Center. The advantage of considering SHADOW is that the code is freely available and is based, for the most part, on tcpdump filters. This discussion takes place in Section 4.4.

Finally, in Section 4.5, we look at using statistics to go beyond simple signature-based intrusion detection.

D. J. Marchette, *Computer Intrusion Detection and Network Monitoring*

4.2 TCPDUMP FILTERS

The tcpdump utility (see Section 1.2) has a built-in capability to filter packets based on the header fields. This allows the user to look for certain known attacks or unusual packets that may be indicative of new attacks, as we will see in the rest of this chapter.

The syntax of the filters is quite simple. It makes use of a number of keywords:

src, dst, host, net, port, ip, tcp, udp, icmp, and, or, not.

In addition, specific fields can be addressed by their positions in the header. There are other keywords, for example those specific to other protocols, but we will focus on these alone. See the man page for more details.

Some examples will make this clear. A filter to select all TCP packets to the machine 10.10.2.7 is simply

```
tcp and dst host 10.10.2.7
```

To select TCP packets to all machines on the 10.10.x.x network we use (note the final period)

```
tcp and dst net 10.10.
```

Parentheses can be used for grouping. Consider the following

```
tcp and ((dst port 22) or (dst port 23)) and host waldo.ourhouse.org
```

This filter looks for TCP packets that are sent to either port 22 (ssh) or 23 (telnet) and are to or from the machine waldo.ourhouse.org. Note that you can specify the machines as either IP addresses or domain names, although specifying domain names requires a DNS lookup initially, which may take a while if the filter contains a large number of hosts.

Specific fields are addressed using a bracket notation

```
ip[6:2] & 0x1fff = 0
```

This tests the 2-byte field at position 6 bytes in the IP header, which contains the flags and fragment-offset fields. It tests to see whether the fragment offset is 0, so we can use this to find the first fragment of a fragmented packet

```
(ip[6:1] & 0x20) and (ip[6:2] & 0x1fff = 0)
```

The first part finds packets with the "more fragments" flag set, and the second ensures that it's the first fragment.

We can look for ranges using this notation as well. To find UDP packets with destination port values less than 20, we use

udp[2:2] < 20

Note that we cannot use something like: "dst port < 20"; we must use the field index notation for ranges and bit manipulations.

We can combine these filter fragments using the "and" and "or" operators and can negate using the "not" keyword. Thus, a filter to detect imap scans could be written as

port 143 and not (dst host imapserv1.ournet.com or dst host imapserv2.ournet.com or src host imapserv3.friendnet.com)

The filters are "compiled" into a table for fast execution. Thus, there is a small initial overhead in parsing the filters, but the execution has been optimized to reduce packet loss. There is, however, a limit to the size of a filter.

4.3 COMMON ATTACKS

4.3.1 DOS Attacks

Denial-of-service (DOS) attacks attempt to shut down a network, computer, or process, or otherwise deny the use of resources or services to the authorized users. Generally speaking, DOS attacks at the network level attempt either to shut down a computer or network, cause a dramatic slowdown in performance, or shut down or otherwise make inaccessible a given service.

The listing in this section is not comprehensive, especially because new attacks are invented almost daily, it seems, but rather gives a flavor for the kinds of attacks that are possible.

4.3.1.1 Land Attack In the land attack, a TCP SYN packet is constructed with the source and destination IP addresses the same and both set to the target machine. On some older systems, this causes the system to lock up, and the machine must be rebooted. A single packet is all that is needed by this attack. The land attack signature is shown in Table 4.1.

A land attack is probably not effective in today's environment, but due to software reuse and potential coding errors in future systems, there is always the possibility that this attack, like any other, may become effective once again against some future system.

The land attack illustrates a common thread in most denial-of-service (and many other) attacks. A strange or "impossible" packet is specially crafted, and some bug or unrecognized feature is exercised by the receipt of the strange packet. These "features" can be detected by perusing the source code of the application or operating system, trial-and-error, logical extension of published standards, and by accident.

Note that in this attack the source destination of the packet has been set by the attacker. This is called "spoofing" and can occur either to hide the identity of the attacker or, as in this case, as a fundamental part of the attack.

Table 4.1 Land attack signature.

Protocol	Specifics	Effect
TCP	SYN packet with same source and destination	Locks up system

Example:
06:49:55.47 10.10.2.23.139 > 10.10.2.23.139: S

Filter: ip[12:4] = ip[16:4]
Comment:
General filter to detect any IP packet with equal source and destination. Note that we cannot say the more natural "src host == dst host."

Table 4.2 Neptune attack signature.

Protocol	Specifics	Effect
TCP	SYN packet from unreachable host	Overflows connection buffer

Example:
09:23:17.47 172.16.43.19.1233 > 10.10.2.23.25: S
09:23:17.61 172.16.43.19.1234 > 10.10.2.23.25: S
09:23:17.96 172.16.43.19.1235 > 10.10.2.23.25: S
etc.

Filter: This cannot be filtered with tcpdump.
Comment:
The signature requires that 172.16.43.19 be unreachable. The destination IP addresses need not be the same so long as they are unreachable. The destination ports can be any open port. The source ports are arbitrary. Some services have mechanisms in place to restrict the number of connections, so this may be ineffective against those services.

4.3.1.2 Neptune Neptune, or "SYN flood," utilizes the fact that for each half-open TCP connection made to a machine, tcpd (the program that handles telnet, FTP, and other connections) creates a record in a data structure to hold the information about the connection. If the connection is not completed within a certain amount of time, the connection "times out" and the record is freed. If

enough connections can be initialized before the timeout occurs, the data structure can overflow, causing a segmentation fault and locking up the computer.

In this attack, the packets are crafted to have a source IP address that is unreachable. This is so that no host responds to the SYN/ACK sent by the target, forcing the connection to stay open. A large number of such SYN packets are sent to the machine in a short amount of time. See Table 4.2 for the Neptune attack signature.

4.3.1.3 Ping O' Death The Ping O' Death is an ICMP echo request (ping) packet with an illegally long (longer than 64K bytes) payload. Older operating systems lock up or reboot when the buffer into which the incoming packet is stored overflows. Early versions of Windows95 had a ping program that would allow one to specify the packet length, even if the length was too big for a normal packet, making this a particularly popular attack for a while. As with the land attack, this requires only a single packet to be effective. Also, like the land attack, few if any modern operating systems are vulnerable. The Ping O' Death signature is shown in Table 4.3.

Table 4.3 Ping O' Death signature.

Protocol	Specifics	Effect
ICMP	Packet larger than maximum IP packet	Locks up system

Example:
172.16.12.37 > 10.10.2.23: icmp: echo request (frag 1213:350@0+)
172.16.12.37 > 10.10.2.23: (frag 1213:350@350+)
172.16.12.37 > 10.10.2.23: (frag 1213:350@700+)
172.16.12.37 > 10.10.2.23: (frag 1213:350@1050+)
.....
172.16.12.37 > 10.10.2.23: (frag 1213:350@65100+)
172.16.12.37 > 10.10.2.23: (frag 1213:300@65450)

Filter: icmp and (ip[6:1] & 0x20 != 0) or (ip[6:2] & 0x1fff != 0)
Comment:
This only detects fragmented ICMP packets. One must then check to see whether the packet is too large.

4.3.1.4 Process Table The process table attack was developed by MIT Lincoln Labs for DARPA to be used as part of a test of intrusion detection systems. The idea was to develop a new attack to see whether the systems would be able to detect it. It is an attack against Unix systems. The basic idea comes from the fact that each time an incoming TCP connection is received, a process is forked. By initiating many connections, the attacker can fill up the process table. Once

the table is full, no new processes can be spawned, so nothing can be done on the computer. See Table 4.4 for the attack signature.

This attack must be mounted against a service that accepts connections but not against one that restricts the number of connections accepted. For example, sendmail will not accept new connections if the load average is too high, so it is not a good target for this attack.

Table 4.4 Process table signature.

Protocol	Specifics	Effect
TCP	Large number of connections initiated	Locks up the system

Example:

07:42:16.57 172.16.43.19.1233 > 10.10.2.23.79: S

07:42:16.64 172.16.43.19.1234 > 10.10.2.23.79: S

07:42:17.06 172.16.43.19.1235 > 10.10.2.23.79: S

.....

Comment:

There is no way to filter or detect this at the single packet level.

One must tally the number of connections between machines.

One example of this attack is to initiate a large number of finger sessions. The finger program is a method of sharing information about users that pre-dates the Web. Some earlier versions of finger did not time out, which means that once you open a connection it stays open until you close it (or the machine is rebooted). Each connection gets its own process ID, and if enough connections are initiated, eventually the process table is full. This is particularly annoying to anyone legitimately using the machine since any command will generate a "no more processes" message and fail to execute.

A related phenomenon is caused by a program using all the machine memory. If a program allocates all the memory that the machine has (including most of the swap space), typing any command (even ls) will cause a core dump (the program crashes). This is usually the result of a programmer not taking care to check the available memory before allocating, but it could also be the result of a worm or other malicious code.

4.3.1.5 Targa3 The Targa3 attack sends a combination of illegal packets to the victim machine. These malformed packets cause some systems to crash, and even those that are not specifically harmed by the packets will use up resources dealing with the packets. These packets have one or more of the following:

- Invalid fragmentation, protocol, packet size, or IP header values;
- Invalid options;

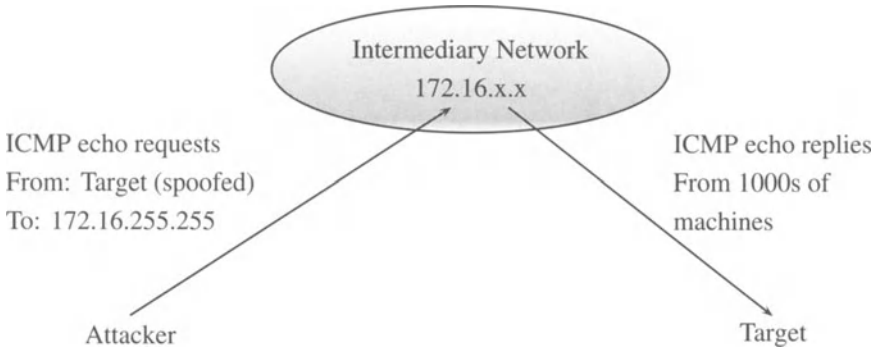

Fig. 4.1 A Smurf attack.

- Invalid TCP segments;
- Invalid routing flags.

Rather than list all possible malformed packets (which would be necessary for a tcpdump filter), the way to detect this attack is simply to check each incoming packet for legitimacy.

4.3.1.6 Smurf Attack The Smurf attack has three participants: the attacker, the target, and an intermediary who is fooled into actually mounting the attack. The attack is depicted in Figure 4.1. The attacker constructs echo request packets (ping) with the target as the source IP and the intermediary as the destination IP. These are broadcast, to maximize the number of machines responding. The machines at the intermediary network all respond to the echo request with packets destined for the target machine. The target machine cannot process the large number of packets received and goes down under the load. At the very least, it is unable to process legitimate connections and so is effectively cut off from the network. The signature is shown in Table 4.5.

Note that, from the target's perspective, the attacker does not appear in the network trace. Only by looking at the intermediary network's logs (if they are kept) can the attacker be traced.

4.3.1.7 Syslogd Attack The syslogd attack kills the syslogd demon on a Solaris server. Older versions of this demon would crash if given a source address with no DNS entry, so the attack consists of packets sent to the syslog port, where the source IP address has been spoofed to be one without a DNS entry. The signature is shown in Table 4.6.

Note that we cannot tell that there is no DNS entry for the source without doing a DNS lookup ourselves. This is unnecessary since none but our own machines should be connecting to our syslog. These should be blocked at the firewall, so the attack should fail on any reasonably secure network even if there are old, unpatched Solaris machines on the network. (Of course, a reasonably secure network should not have unpatched machines on it in the first place.)

Table 4.5 Smurf signature.

Protocol	Specifics	Effect
ICMP	Echo requests sent to broadcast with the target host spoofed as the destination	Target taken off the network

Filter: icmp and (ip[19] = 255) or (ip[19] = 0)
Comment:
This only detects attempts to use your network as an intermediary. Filtering for ICMP echo replies combined with further processing to count the number of packets to any individual machine is necessary to detect a Smurf attack. A solution is to deny ICMP echo replies at the firewall unless they are in response to an outgoing ICMP echo request. This requires a stateful firewall. Some people advise blocking (nearly) all ICMP packets at the firewall.

4.3.1.8 Teardrop Teardrop takes advantage of the fact that some older TCP/IP implementations do not properly handle overlapping fragments. An attacker sends a series of packets carefully crafted to look like a normal packet that has been fragmented but such that the fragments overlap instead of being disjoint. The receiving machine crashes. An example is given in Table 4.7.

4.3.1.9 UDP Storm UDP storm causes two of your machines to attack each other. The idea is that there are a number of ports that will respond with another

Table 4.6 Syslogd signature.

Protocol	Specifics	Effect
TCP	Source host has no DNS entry	syslogd crashes

Example:
11:23:17.42 172.16.51.2137 > 10.10.13.32.514 S:

Filter: tcp and (dst port 514) and not (src net 10.10.)
Comment:
Look for external connections to the syslogd port.

Table 4.7 Teardrop signature.

Protocol	Specifics	Effect
UDP	Fragmented packet, fragments overlap	Locks up system

Example:
07:21:33.21 172.16.123.37.23453 > 10.10.2.23.53: udp (frag 1213:350@0+)
07:21:33.21 172.16.123.37.23453 > 10.10.2.23.53: (frag 1213:300@350)

Filter: udp and (ip[6:1] & 0x20 != 0)
Comment:
This only detects fragmented UDP packets.
One must then check to see whether the fragments overlap.

packet if a packet is sent. Echo (port 7) and chargen (port 19) are this way. Echo will echo the packet back, while chargen will generate a stream of characters.

Consider a UDP packet with source port 7 and destination port 19. The packet generates some characters from the destination machine, headed for the echo port of the source machine. The source machine echoes these packets back, generating even more packets, and so on. Eventually, both machines are spending all their time sending packets back and forth until one or both of them go down. See Table 4.8 for the signature.

As a matter of course packets that come from outside your network that have a source IP address inside your network should be blocked at the firewall. These

Table 4.8 UDP storm signature.

Protocol	Specifics	Effect
UDP	Source Port: 7 Destination Port: 19	Both hosts lock up

Example:
11:23:17.42 10.10.2.34.7 > 10.10.2.37.19

Filter: udp and (src port 7) and (dst port 19)
Comment:
Other ports can be used, as long as they both respond to packets.

are almost certainly spoofed packets. Even if they are not an attack they are an indication of something gone wrong.

4.3.2 Probes and Network Mapping

One of the first things an attacker needs to do is determine information about the hosts on your network. This includes a list of the IP addresses that are valid and the operating systems and services running on the hosts. Network mapping refers to the act of obtaining a list of the hosts on the network, whereas probing refers to methods for determining specific information about individual machines.

A typical attack involves first mapping the network to determine the active machines, followed by probing select machines to determine the operating system and services running on the machine, selecting a service for which a known vulnerability exists, and launching an attack against the selected machine and service. We will discuss each of these topics in turn.

4.3.2.1 Network Mapping The simplest network mapping technique is to send a ping to broadcast and see who replies. If the target network is the class B network 10.10.x.x, the attacker sends some packets such as

11:42:16.33 attacker.com > 10.10.255.255 ICMP: Echo Request

Several packets are sent to ensure that the information is not corrupted by lost packets. The source address cannot be spoofed since the return packets must be examined. However, the attacker can send several packets with spoofed addresses along with the real ones in order to sow confusion. This kind of mapping attack is trivial to detect and can easily be blocked by a firewall that refuses to pass broadcast packets.

Another approach is to send packets to every possible computer on the network. Using TCP, one selects a port that is passed by the firewall, and hence will get to the target machines, and then starts sending packets to each possible IP address. Suppose the firewall allows telnet (port 23) access from any IP address outside the network. Then, the first few packets in this scan look like:

11:47:34.09 attacker.com.2213 > 10.10.1.1.23 S
11:47:34.19 attacker.com.2213 > 10.10.1.2.23 S
11:47:34.27 attacker.com.2213 > 10.10.1.3.23 S
etc.

Again, this is easy to detect if the attacker is this single-minded. If the target machines are randomized, however, and the packets are spread out in time (a so-called low-and-slow scan), this kind of attack can be difficult to detect. There is a tradeoff of course. With over 65,000 possible machines in this network, sending one packet a minute results in a scan that takes a month and a half. If the attacker has the patience to do this, the attack may go undetected. However, it only takes one alert system administrator to see the connection attempt and become curious

and detect the attack. Also, one packet a minute might be too many; the attacker may want to send only one an hour, in which case this scan is not feasible.

A more sophisticated attacker might try to target the scan more intelligently. For example, it may be enough for the attacker's purposes to find a small number of machines, in which case a randomized low-and-slow scan as above will probably be successful. It stops when enough machines have been detected.

There are ways to make the scan harder to detect. The preceding scan used SYN packets, which are logged on many machines and which most intrusion detection systems watch. By simply changing the flag, one can make the scan harder to detect:

10:47:34.33 attacker.com.2213 > 10.10.17.121.23 S ack
11:13:21.24 attacker.com.2213 > 10.10.3.207.23 S ack
12:11:11.53 attacker.com.2213 > 10.10.51.14.23 S ack
etc.

In this case, the scan looks as if the attacker is simply responding to a series of connection requests from machines on the target network. If the attacker is slightly more clever, the scan can look as though the target machines are simply Web surfing (port 80):

10:47:34.33 attacker.com.80 > 10.10.17.121.2214 S ack
11:13:21.24 attacker.com.80 > 10.10.3.207.2043 S ack
12:11:11.53 attacker.com.80 > 10.10.51.14.3219 S ack
etc.

In order to detect this kind of scan, the intrusion detection system must either be stateful (remember that there were no outgoing SYN packets to initiate the sessions) or keep a (current) list of all active machines on the network.

Another option is to send reset packets:

11:47:34.09 attacker.com.2213 > 10.10.17.121.23 R
11:53:43.12 attacker.com.2213 > 10.10.3.207.23 R
12:31:24.01 attacker.com.2213 > 10.10.51.14.23 R
etc.

Reset flags are used to indicate that something has gone wrong with a connection session and are passed by most firewalls (stateful firewalls can detect that the reset was not the result of an ongoing connection and deny it.) They are not logged by many hosts, and they are common enough that they are often ignored by intrusion detection systems. Again, unless the system is stateful, these have a very good chance of going undetected.

Several examples of reset scans are discussed in Green et al. [1999]. Resets can be explained in a number of ways:

- As discussed earlier, a reset is sent during the normal functioning of TCP/IP. For example, if a machine sends a SYN packet to a port that is not accepting

connections, a reset packet will result. Thus, an incoming reset that is in response to an outbound connection attempt is not an indication of a reset scan.

- Incoming resets can be the result of an attack on a third party. As in the Smurf attack, if an attacker sends packets to a network with the source address spoofed, all the response packets (SYN/ACKs or resets, for example) go to the spoofed machines and can show up as if it were a coordinated scanning or denial-of-service attempt.
- Reset scanning is an inverse mapping technique. If a machine receives a reset packet, it simply ignores it, sending no response. If, however, a router receives a reset packet destined for a machine that does not exist, it will send an ICMP error message indicating that the host does not exist. Thus, the inverse of the machines that generated a response is a list of the machines that are alive on the network.
- A final use for resets as an attack tool is in TCP hijacking. To close off one side of a connection during a hijacking, a reset packet is sent. This is discussed in more detail in Section 4.3.3.2.

Another use for network scanning is to look for installed trojans. Trojans are programs that are loaded on a system (often unknowingly by the legitimate users of the system) that have a sinister hidden purpose as well as their apparent one. The name comes from the Greek horse of the same name, of course. Some trojans listen at particular ports and, upon receipt of a packet to that port, announce their presence to the attacker. The attacker can then make use of the trojan to gain access to the machine. In some cases, the trojan gives the attacker complete control over the machine, even to the extent of turning on the microphone (if one is plugged in) and listening to conversations around the machine! We will discuss trojans in more detail in Chapter 7.

4.3.2.2 Fingerprinting Fingerprinting is the term used for determining the operating system (or other unique identifier) for a system. For example, as we will see in Sections 4.9.1 and 4.9.2, there are programs that will perform this function for you, either actively (by sending packets to the machine) or passively (by analyzing the packets sent to your machine).

As we will see later, fingerprinting can be done in a fairly deterministic manner. Different operating systems react differently to different stimuli, and their reactions can be used to do a pretty good job of operating system (OS) identification.

However, this is not perfect, which means that there is a great opportunity for statistics to play a part. In this section, I will present some thoughts on this topic, although to my knowledge no one has attempted to apply statistical methods to the problem of (passive or active) operating system determination.

Fyodor [1999] is one of the first papers to discuss operating system fingerprinting. Let us consider some of the issues discussed in this paper, with a view to discovering areas in which the statistician may participate.

Fyodor lists a number of techniques for OS determination. He is primarily interested in this paper in active fingerprinting, in which responses to crafted

packets are used to make inferences about the OS of the probed machine. One obvious use for statistics would be to determine the best order of packets, so that we can reduce the number of packets sent prior to making an OS determination. This would be a useful contribution.

One basic technique in OS fingerprinting is to send a packet with a strange flag combination. For example, an unexpected FIN packet will cause some systems to respond even though the "correct" action is to ignore the packet. Similarly, different OSs will respond to strange flag combinations differently. Fyodor reports that early Linux implementations will leave the strange flags set in their response back. Thus, by looking at the response to these unusual flag combinations, one obtains information that can help to identify the operating system of the machine of interest.

Another indicator that can be used is the pattern of sequence numbers chosen by the host. Is it random or deterministic? Fyodor reports that statistics computed on the sequence numbers (such as variance) can be be used to cluster operating systems. This requires a fair number of packets and so might be better as a passive fingerprinting technique.

Many operating systems set the "don't fragment" bit as a matter of course, whereas others only do it under certain circumstances. Newer operating systems tend to set it more often than older ones. This is my favorite example because it allows the clustering of operating systems by considering a single bit (although obviously there are only two clusters). This could be used in either an active or passive fingerprinting system.

There are various options available to an operating system for reporting errors via ICMP. For example, if a machine receives a large number of packets to a closed port, it need not generate an ICMP destination unreachable message for every one. Different operating systems make different choices about what to do in these optional situations.

One of the richest areas for fingerprinting features is the options field. Obviously, since these are optional, operating systems are free to implement those they wish, and to use these pretty much whenever they wish. Thus, the pattern of their usage can be very useful in determining the OS.

Fyodor notes that a certain operating system (which will remain nameless) has not (apparently) changed the stack since 1995 (this is a hint). Thus, it is difficult to distinguish which version of the operating system the host is running, since they all act the same. One method (suggested tongue-in-cheek by Fyodor) is to start with old attacks against this operating system suite and work your way up the list until one of them succeeds. This is one reason why I prefer passive fingerprinting to active fingerprinting: the temptation to be bad can be quite strong.

Now let us focus on passive fingerprinting. A good place to start learning about passive fingerprinting is Spitzner [2000]. Much of the preceding discussion is also relevant to passive fingerprinting. There are several issues unique to passive fingerprinting, however. We will look at these to search for potential applications of statistical methods.

Passive fingerprinting, like active fingerprinting, uses the values of the header fields to guess the operating system of the originating machine. Examples of useful fields are the ones that allow for variability among operating systems, such

as TTL, window size, type of service, and whether the don't fragment flag is set. I conjecture that the value in the urgent pointer field is a potential key to operating systems if the urgent flag is not set.

The first field mentioned, TTL, already points out an area worthy of investigation by the statistician. Since the initial value of this field is unknown (but is the value needed for OS fingerprinting), one must estimate it. One method is to assume the original value to be the smallest power of 2 larger than the current value, or 255, whichever is smaller. An alternative would be to use other values to get a list of tentative OSs and then, using the default values of the TTL for them, determine the best fit for the TTL. Since not all OSs set the value to a power of 2, the second method is more likely to work than the first.

The report by the Swiss Academic & Research Network [1999] gives a listing of the default values of the TTL for various operating systems. They make two observations up front. First, the guidelines for Internet hosts state that the default TTL must be configurable, which means that a sufficiently sophisticated attacker can change the default TTL (or the TTL value of any individual outgoing packet). Recall that this ability is critical to the functioning of traceroute (Section 1.9.5). Second, the default number must be larger than the diameter of the Internet. This is defined as the longest possible path between hosts (recall that loops are not allowed, so this is well-defined). Obviously, this brings up the interesting question of how one estimates the diameter of the Internet.

The values of default TTLs for TCP packets given in Swiss Academic & Research Network [1999] are: 30, 32, 60, 64, 128, and 255. Thus, given a packet with a TTL value of X, one must first determine whether it was originally set at one of these values or another value (which in itself provides information about the attacker) and, if so, which one. This seems to be an interesting problem to investigate.

Note that if we take the advice given by the RFC (which is to set the value at twice the diameter of the Internet) and ignore the 255 value, we have the vendor estimates of the diameter of the Internet at (at least) 15, 16, 60, 32 and 64.

Current methods for passive fingerprinting rely mostly on table lookups. The preceding discussion argues for a tree-based method. Given certain values of the parameters, one may impute the original value of the TTL and, using this estimate and the other values, make a decision. Since each field splits the possible operating systems into groups based on the value, a technique such as CART (Brieman et al. [1998]) seems like an obvious method to try.

4.3.2.3 Probes Several services have known vulnerabilities that can be exploited by an attacker to gain access to the machine. Different versions of the software will of course contain different vulnerabilities, so it is important to determine not only which services the target machine is running but also which versions of the different services are running.

First, the attacker determines the services running by a port scan of the target machine. The simplest is once again to scan all the ports, an approach that is easy to detect. Smarter attackers will only probe for ports that they know they can compromise:

13:12:22.33 attacker.com.2113 > 10.10.17.121.telnet S
13:13:22.54 attacker.com.2114 > 10.10.17.121.smtp S
13:13:22.73 attacker.com.2115 > 10.10.17.121.finger S
13:13:22.97 attacker.com.2116 > 10.10.17.121.http S
13:13:23.13 attacker.com.2117 > 10.10.17.121.imap S
13:13:23.27 attacker.com.2118 > 10.10.17.121.rlogin S
13:13:23.41 attacker.com.2119 > 10.10.17.121.printer S

Once a service has been detected, the easiest way to determine what version it is running is to look at the response it sent. For example, if one types the command

telnet mycomputer.com 25

one gets an answer like

220 mycomputer.com ESMTP Sendmail 8.9.3/8.9.3; Sat, 8 Jan 2000 12:04:01 -0500

Typing help at this prompt results in

```
214-This is Sendmail version 8.9.3
214-Topics:
214-HELO EHLO MAIL RCPT DATA
214-RSET NOOP QUIT HELP VRFY
214-EXPN VERB ETRN DSN
214-For more info use "HELP <topic>".
214-To report bugs in the implementation send email to
214-sendmail-bugs@sendmail.org.
214-For local information send email to Postmaster at
214-    your site.
214 End of HELP info
```

In this manner, many of the services will tell the attacker what version of the software is running and even provide simple help menus for the novice attacker.

Most intrusion detection systems have a list of ports that are considered "bad" and will result in an alert if a packet is sent to one of these ports. An example list is shown in Table 4.9.

Of course, one may want to run some of these services on some of the machines on one's network, so a single global list like this is probably inappropriate for most real networks. Instead, one might add a list of the machines that are running the services listed plus a list of the machines (or networks) that are allowed to access those services.

4.3.3 Gaining Access

Most of the techniques for gaining access to machines are better discussed in the chapter on host monitoring, Chapter 5. However, there are some that can be detected via network monitoring, and we will discuss these here.

Table 4.9 An example "bad ports" list.

Port Number	Protocol	Service	Comment
<20	TCP,UDP		Low-numbered ports
23	TCP,UDP	Telnet	
25	TCP,UDP	SMTP Email	Many vulnerabilities
53	TCP	DNS	Zone transfer
79	TCP,UDP	Finger	Vulnerabilities
111	TCP,UDP	sunrpc	Vulnerabilities
143	TCP,UDP	IMAP	Vulnerabilities
666	TCP,UDP	Doom	Networked game

4.3.3.1 Password Guessing Most machines allow some form of remote access, such as telnet, rlogin, FTP, ssh,. These require a user name and a password. User names are relatively easy to obtain. Many systems use the user name as the email address, so one can obtain user names from email, Usenet postings, or by doing email searches on the Web. Also, most operating systems have specific user names for particular tasks: root, lp, admin, guest, and so forth.

Once one has a user name and has determined that a service is running (such as telnet) to allow remote access, one can attempt to log in as that user. If the system is poorly maintained, some of the accounts may not even require a password (in the past, Silicon Graphics machines were shipped with no password for the user "lp"). Otherwise, one can try to guess the password.

From the perspective of network monitoring, password guessing shows up as a large number of connections to telnet (or whichever service is utilized) that end abruptly. If one looks at content, these are obvious from the number of different passwords attempted. Of course, people are often forgetting their passwords, and so this could simply be the legitimate user trying likely passwords. However, any such attempts should be investigated.

It should be noted that in spite of the movie portrayals, guessing passwords in this manner is particularly ineffective. A better way to determine a password is either through social engineering or by obtaining the password file and running a password cracking program on it.

Social engineering is one of the most useful tools for a hacker intent on unauthorized access. For example, the user receives a call from someone claiming to be a representative of the manufacturer of the computer. It seems that there is a potential problem with one of the system files on the machine, and the company engineer needs access to an account so that it can be checked and, if necessary, fixed. This is all part of the maintenance agreement. Or the caller is a harried vice president who can't remember his password and needs to get his viewgraphs off the company server. If he can just get access to the machine for a few minutes he can download his viewgraphs to his laptop. Surprisingly, there are people who will dutifully provide the caller with an account on the machine.

4.3.3.2 TCP Hijacking TCP hijacking is a clever attack that takes advantage of the fact that computers generally only check things once. It also takes advantage of the fact that many (most?) operating systems are not written with security foremost in mind. This is the attack that Kevin Mitnick used against Tsutomu Shimomura's system, as described in Northcutt [1999]. See also Hafner and Markoff [1995].

Consider Figure 4.2. The attacker wishes access to computer A. The attacker knows that computer A trusts computer B. This has been determined by some intelligence gathering described in Northcutt [1999], pages 5–7. Recall how a connection is set up (see Section 1.5.4.1, Figure 1.8). A SYN packet is sent with a unique sequence number from B to A. A responds with a SYN/ACK and a unique acknowledgment number. B responds with an ACK. The idea of TCP hijacking is to pretend to be the trusted machine B, set up a connection, but have the connection be between A and the attacker rather than between A and B. In order to accomplish this, the attacker must perform the following steps.

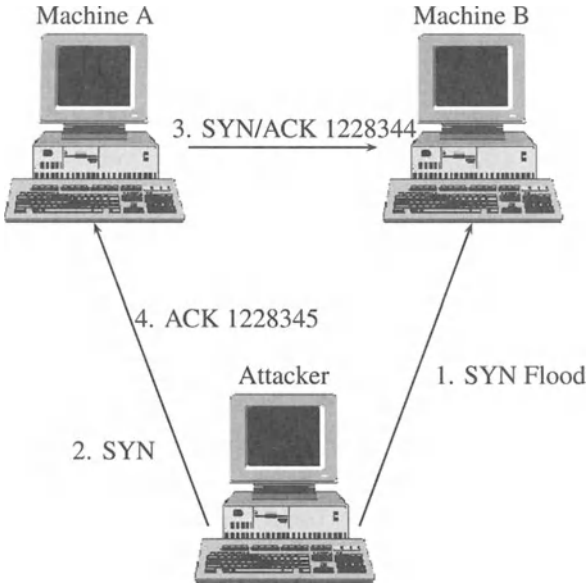

Fig. 4.2 TCP hijacking. 1. Attacker SYN floods machine B to make sure it does not respond to any packets from machine A. 2. Attacker initiates a connection with Machine A using a SYN packet spoofed to appear to be from Machine B. 3. Machine A acknowledges the connection. 4. Attacker sends an ACK packet with the correct sequence number to Machine A, finishing the three-way handshake. This assumes that the attacker has previously determined the sequence number algorithm that A uses, and has determined the next sequence number that A will use.

1. Determine the next sequence number to be used by A.
2. Take B off the network so that it cannot respond.
3. Send a SYN packet to A with B as the source.

4. Send an ACK packet to A using the acknowledgment number A expects, now with the source host as the attacker.

The sequence number rule is determined by sending a series of connection requests to A and analyzing the sequence numbers with which it responds. Often, these use a simple algorithm such as adding a constant to the last number used. Once the rule has been determined, a SYN packet followed by a RESET is sent to determine the current sequence number. Then, the SYN packet is sent to initiate the connection.

Machine B can be taken off the network with a SYN flood or similar denial-of-service attack (Section 4.3.1.2). With B unable to respond to A's SYN/ACK, the attacker is free to jump in with the right response to the SYN/ACK, now with the attacking machine as the source IP address, and the connection is established. Further packets now proceed between A and the attacker.

Why isn't the change in IP addresses detected? After all, the final ACK is coming from a different machine than the original SYN. Shouldn't this be noticed? The reason it generally is not noticed can be placed on the doorstep of the layered approach to networking. The IP address is at the IP layer, while the sequence number, which determines the connection to which the packet belongs, is handled at the TCP (protocol) layer. Thus, unless the TCP layer specifically looks back at the IP header to validate the connection, it does not keep track of the IP addresses at all. Similarly, since the IP layer does not handle the connections, it does not know to check that the IP addresses have changed. As far as it's concerned, these are just more packets to be forwarded to the protocol layer.

Furthermore, A trusts the attacking machine. The first packet, which had B as the source address, was used to determine whether the machine asking for the connection is trusted. Once it was determined that B is a trusted machine, the connection is deemed to be with a trusted machine, and since the change in IP addresses is not noted, the attacking machine inherits this trust relationship.

Hijacking is easy to detect and foil. A stateful firewall can notice that the IP addresses have changed and disallow the connection, and network monitors can watch for connections that change IP addresses in midstream. Also, hijacking can be discouraged by making sequence numbers difficult to guess. For example, Linux uses a random number generator to generate new sequence numbers, making it extremely difficult to hijack sessions in this manner.

4.4 SHADOW

SHADOW, which stands for Secondary Heuristic Analysis for Defensive Online Warfare, is a project developed at the Naval Surface Warfare Center (NSWC) for the purpose of detecting intrusion attempts into the network and correlating data across multiple networks.

SHADOW is a suite of freely available software consisting of tcpdump filters, perl scripts, and Web pages for the detection and display of unusual or inappropriate packets. The system is designed to be configured to the network in order to allow security people to tune the filters as appropriate to their network. The

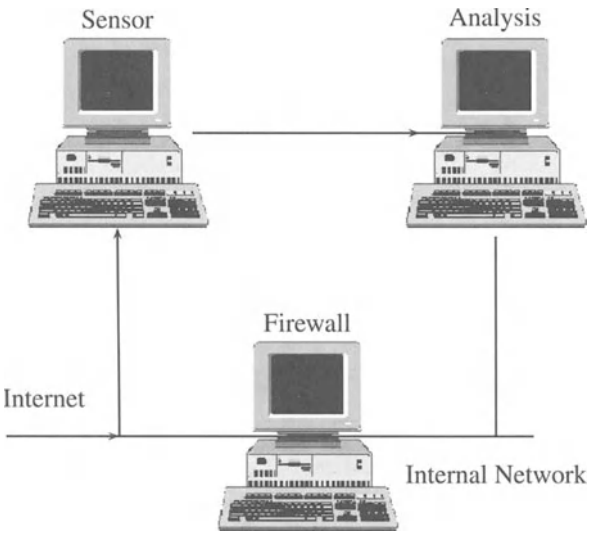

Fig. 4.3 A typical SHADOW configuration. The sensor is passive with no IP address for the network card on the Internet. Special care is taken to ensure that no packets are sent out onto the external network by the sensor and that only connections from the analysis station, on the internal network card, are allowed.

software also helps the security analyst to write and send intrusion reports, gather information on the intruder (through nslookup (Section 1.9.2) and whois (Section 1.9.3) commands), and gather information on the security of the protected network (using nmap (Section 4.9.1)). The software allows the monitoring of multiple sites and allows simple searches of the data to determine the context in which a suspected attack occurs or to determine the past activity of a suspected attacker.

SHADOW is an "interval-based IDS," rather than a "real-time IDS." This means that the data are collected over a period of time (usually an hour) and processed in batches rather than having the system on-line, detecting attacks as they happen.

The usual SHADOW setup (Figure 4.3) is to have two machines dedicated to SHADOW. The first, called the sensor, is a machine (usually a Linux box with two network cards) sitting outside the firewall. The second, called the analysis station, is usually a high-end Linux box, which resides inside the firewall. The sensor does little more than collect packets. On an hourly basis, it transmits (via secure shell) the last hour's worth of data to the analysis station for processing. Actually, for obvious reasons, the analysis station initiates the transfer, getting the data from the sensor rather than the sensor sending data to the analysis station unasked.

The reason the sensor sits outside the firewall is to ensure that all attacks (even those that do not succeed in passing the firewall) are detected. Although firewall logs could be used to detect unsuccessful attacks, they are (surprisingly) not always available to the people doing intrusion detection. The people running the network (and hence the firewall) are not always the people in charge of security, even

network security. This seems counter-intuitive (and it is), but is not that uncommon. It clearly is not the best model for security.

Some people advocate having a second sensor inside the firewall. This allows the security officer to determine which packets passed the firewall and which were knocked down (by differencing the inside and outside files). It also gives a second line of defense, protected by the firewall, in the event that the outside sensor is taken out.

There are some differences of opinion about how the sensor should be configured. One model is to have a single network card and have all communication with the sensor go through the firewall. This has the disadvantage that it increases the load on the network (as the packets are sent to the analysis station). It also means that the sensor is visible to the outside since it needs to have an IP address.

An alternative is to give the sensor two network cards. The first, connected to the incoming network, is a read-only interface (made so with software, or by a judicious use of wire cutters). This is the sensor card, which watches the traffic and makes copies of all the packets. This card does not have an IP address associated with it, so the sensor is relatively invisible to the outside world. Since it is read-only, the sensor cannot be used to send information out even if it is compromised. The second card is connected to a local area network inside the firewall, which also serves the analysis station. In this manner, the data transfers do not impact on the protected network, and the sensor is protected by the firewall.

The downside to the preceding model is the extra hardware involved (not a big burden), and the fact that the local area network does provide a passage around the firewall. This passage is not easily accessible from the outside since the read-only interface and the lack of an IP address make this a difficult target to exploit, yet there is at least a theoretical possibility of data (for example, malicious code) being transferred inside without going through the firewall.

After collecting an hour's worth of data, the packets (in tcpdump binary format) are compressed (using gzip, a public domain compression utility) and sent to the analysis station via secure shell. Thus, the analysis station works on files in one hour increments. The collection times overlap slightly to ensure that no packets are missed during the transition between the hourly data collections.

The analysis station processes the file by first uncompressing it and running its tcpdump filters on the packets. Those packets that pass the filters are considered "suspicious" and will be displayed on a Web page. Other statistics, such as the number of different machines a host tried to access, are calculated, and those above a (user settable) threshold are also added to the Web page.

The suspicious packets are then sorted by outside IP address, and all the suspicious packets from a given IP address are shown together (in time order) on the Web page. The analyst can then look at each hour's worth of data, decide which constitute attacks worth reporting, and generate a report containing the suspicious packets, and host identity information.

Multiple sensors can report to a single SHADOW analysis station. This can be used to monitor the traffic inside one's network to watch for insider attacks. It can also be used to implement enterprise-wide monitoring. I am aware of several organizations that use SHADOW to monitor several sites, utilizing one to two

analysts for the monitoring. The data for each site is transferred to the central headquarters where the SHADOW filters are applied.

Doing all the monitoring at a central site works well if the monitored sites do not have too much traffic and are relatively homogeneous. It makes sense for larger organizations to have an analyst at each site, with only intrusion reports forwarded to the central site. These choices are left entirely up to the organization using the software.

One of the philosophies behind SHADOW is that the analyst needs more information than a flashing red light saying an intrusion has been detected. SHADOW provides this information by allowing the user to pull up all the data relevant to a particular suspicious event and even to search past data for similar activity. This allows the analyst to do a better job of determining the true nature of the suspicious event.

Another side of this philosophy is that events that are probably not attacks but might be precursors, or simply of interest to the user, are reported. For example, a site may want to see any traceroute (Section 1.9.5) attempts to the site because this is a standard information gathering technique. Although a single traceroute is in and of itself not an attack, it can provide a heads-up to alert the security officer to watch for future activity from the source network.

In addition to monitoring, since SHADOW keeps all the packet headers, the data can be archived to allow historical searches and statistical analysis. Since the data can be quite large, it is a challenge to maintain a database with more than a few month's worth of data unless one uses data reduction methods.

To get an idea of the magnitude of such project here are some statistics. A typical day's worth of data at NSWC is about 2 Gigabytes (compressed). The file for 11AM–12 Noon, April 3, 2000 was 22 Megabytes (compressed) consisting of just under a million packets. It is important to note that this is a peak time, early morning and late night traffic is much less.

SHADOW does not search the content of the packets, so only the headers are stored. This is adjustable, up to full content, if desired. There were scripts originally for searching content, but these are not a part of the SHADOW distribution. Keep the preceding discussion about data sizes in mind before you consider going to full content.

A possible extension to SHADOW would be to add snort (Section 1.9.7) to allow content searching and more specific signatures through its more extensible filtering capability. Since snort uses tcpdump binary format files just like SHADOW, it is a simple matter to add this functionality in an ad hoc manner if desirable.

4.5 ACTIVITY PROFILING

Profiling the activity on the network is the act of collecting statistics that give a summary of the kinds of activities that are naturally occurring on the network. This gives a picture of the normal traffic on the network, which can be used to detect intrusions as deviations from this normal behavior. It can also be used to get a better understanding of the machines on the network by clustering machines

into specific activity clusters. We will look at several ways of constructing these profiles.

First, one must know what services are running on the different machines on the network and to what extent each machine is accessed through the various ports available. One could collect these data by interviewing the system administrator for each machine or by requiring this information as a condition of operation, or one can probe each machine with a scanner such as nmap (see Section 4.9.1) to determine which ports respond to access attempts.

Another approach is to consider the services running as they are represented in the network traffic. For example, to determine which services are running under TCP on each machine, and their relative levels of activity, we could tally the number of SYN packets sent to each port, keeping a separate tally for each port. This represents the activity levels of services that are requested from outside the network. Alternatively, we could tally the SYN/ACKs from our network to the outside, which represents the subset of those services requested that are actually available on the machine. Finally, we could consider the outgoing SYN packets for each destination port, which represents the kind of services the machine typically attempts to access from other machines.

These three tallies represent the activity of the machine. We can cluster machines based on these activity vectors in a number of ways. It seems reasonable to cluster the machines by each activity vector individually, so that one gets clusters for each of the three types of activity of interest. Alternatively, one can cluster the machines by the combined activity represented by the three vectors.

Once we have the activity profiles for the machines, we can look for deviations from these "normal" activity profiles. Intrusions can be detected by looking for activity that has not been seen before on a given machine or activity levels that are greater than is normal. Thus, activity profiles give a way of detecting possible intrusions by the detection of outliers.

If we also keep track of which machines normally interact with a given machine, we can detect when new machines attempt access, which can be used as an indicator of possible attacks.

With machines clustered by activity level, a different kind of outlier can be investigated: those machines that do not fit well into any cluster. These are the machines that are unusual when compared to the rest of the machines on the network. This information can be used to determine which machines require specialized attention. It can also be used to detect machines that may have security holes, such as trojan programs installed on them.

We will consider an activity profile for a given machine (or cluster of machines) to be a vector of counts or probabilities. Each count is associated with a specific activity, such as TCP SYN packets sent to a specific port.

There is an important issue to address. How should the vectors be collected? Should one simply average the number of activities per hour without regard to which hour of the day or night it is? This does not seem to be the best way to approach this problem. An alternative would be to keep an individual tally for each hour of the day. This results in multivariate data, which could be viewed as functional data. If one also makes a distinction between the days of the week, the activity vector for a given machine/port pairing can be thought of as a function

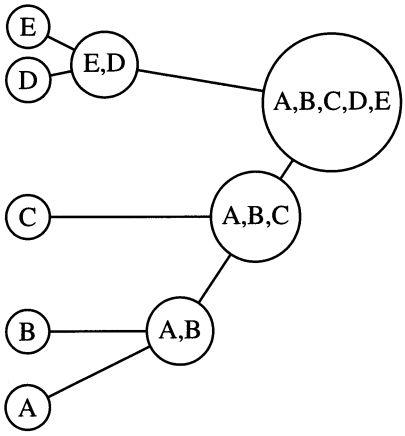

Fig. 4.4 An illustration of hierarchical clustering.

indicating the average number of packets within a given time period (say one hour) at any given time of the week.

4.5.1 Clustering by Activity Level

Clustering is an inherently difficult problem, due in part to the difficulty of defining clusters. Different algorithms are appropriate to different definitions of a cluster, and it is not always apparent what definition is appropriate for a given problem. A good reference for clustering algorithms is Everitt [1993].

Several different clustering algorithms will be described later. Here we will describe one of the most common techniques, hierarchical clustering.

Hierarchical clustering has two main variants, agglomerative and divisive. We will only consider the agglomerative method here, see Everitt [1993] for more information on this and other clustering methods. The idea of agglomerative clustering is initially to place each observation in its own cluster. Subsequently, the two closest clusters are merged, and this is repeated until there is a single cluster. This is illustrated in Figure 4.4.

To decide which clusters to merge, the distance between clusters is computed, and the two clusters with the smallest distance are merged. Different definitions of cluster distance result in different properties of the clustering. Three common definitions are illustrated in Figure 4.5. In complete linkage clustering, the distance between two clusters is taken to be the distance between their furthest points. This results in clusters that are tight since large clusters are necessarily far from all other clusters. With nearest-neighbor clustering (the distance used in the approximate distance clustering method described later), the distance is taken to be the minimum distance between points in the clusters. Other possible distances are the group average (compute the average distance between points), or the distance between the means (possibly scaled by inter-group covariance).

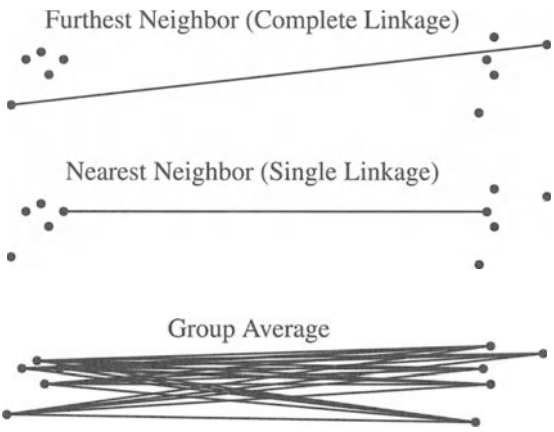

Fig. 4.5 An illustration of distances used in clustering.

Hierarchical clustering results in a tree of clusters, from the root, consisting of the cluster of all the data, to the leaves, consisting of a separate cluster for every observation (see Figure 4.7). This presents one with the problem of deciding the proper set of clusters for the data. This is not unique to hierarchical methods. In fact, all clustering methodologies require some way to determine the number of clusters to use and validate the clusters chosen.

As we will see later, many of the most useful techniques are subjective, utilizing some clever visualization technique to depict the clusters and then relying on the training of the data analyst to decide whether the clusters are well-chosen or not. There are also a number of quantitative techniques available.

If one has the luxury of using tagged data, where each observation has a tag (class label) associated with it, one can validate the clusters by determining their "class purity." For example, suppose one had data measured from K users, where each observation consisted of measurements taken from one session (for example, keystroke timings on a password). After clustering the data (without using the user information in the clustering), one could assign to each cluster a class label (based on the observations clustered in the class) and a purity based on the percentage of observations whose associated user matched the cluster label.

One generally does not have pre-classified data of this sort when tackling a clustering problem. Usually, the whole point is that there is no a priori information about the clusters, so one needs a method for validating the clusters without this extra information. This requires some measure of cluster "goodness," which is then computed for the clusters at hand.

For example, generally one thinks of clusters as being groups of observations that are distinguished from other groups by being closer to each other than they are to the other groups. Figure 4.6 depicts data grouped into three clusters, indicated by the different plotting symbols. Most people would agree that these data are correctly clustered (although some might argue the point). These clusters have the property that for each cluster the within-cluster variance is smaller than the

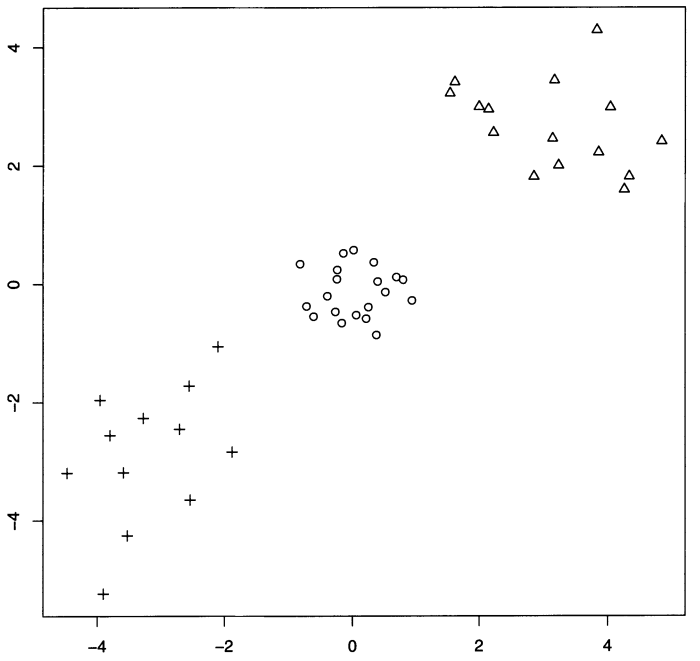

Fig. 4.6 An example of simple clusters.

between-cluster variance, which is a measure of cluster purity for symmetric ("ball-like") clusters such as these. These clusters are easy to distinguish, both by the eye and using clustering algorithms.

4.5.2 Visualizing Clusters

Given data that have been clustered, we want a method for deciding whether the clustering algorithm has produced a "good" set of clusters. There are a number of quantitative measures of goodness that can be used (see any book on clustering, for example Everitt [1993]) but it is important to be able to look at the data and the cluster structure and assess by eye whether the clusters are appropriate. This is not a substitute for the quantitative assessors since the eye can be fooled, particularly with high-dimensional data, but it is a useful addition. However, as we have seen, visualizing high-dimensional data is quite difficult. In this section, we will consider methods for visualization that are appropriate for visualizing cluster structure.

4.5.2.1 Dendograms A dendogram is a plot depicting the tree structure of a hierarchical clustering algorithm. The tree depicted in Figure 4.4 depicts a small

dendogram. The branches connect the sets to those in which they are subsequently grouped.

Another example is provided in Figure 4.7. This depicts a famous data set that relates certain measurements on flowers to the species of the plant. Three different species are represented in the data. Starting at the top of the figure, we can see that if one wishes to cluster these data into two clusters, the data will be split roughly in two, with Setosa in one cluster, Virginica in the other, and Versicolor split between the two clusters. By traversing the tree, we can infer quite a bit about the cluster structure of the data, even though the data may be too high-dimensional to be conveniently investigated through scatter plots (in this case, the data are four-dimensional).

4.5.2.2 Color Histograms and Data Images It is difficult to display high-dimensional data in a manner that is readily interpretable to humans. Several approaches have been suggested, including pairs plots (Section 2.5.2), parallel coordinates (Section 2.5.3), and color histograms (also called "data images"). Some techniques are described in Solka et al. [2000].

For our purposes, we will use the terminology "color histogram" and "data image" fairly interchangeably, but there is a distinction that can be made. In essence, the color histogram is an image of the data, with the only processing

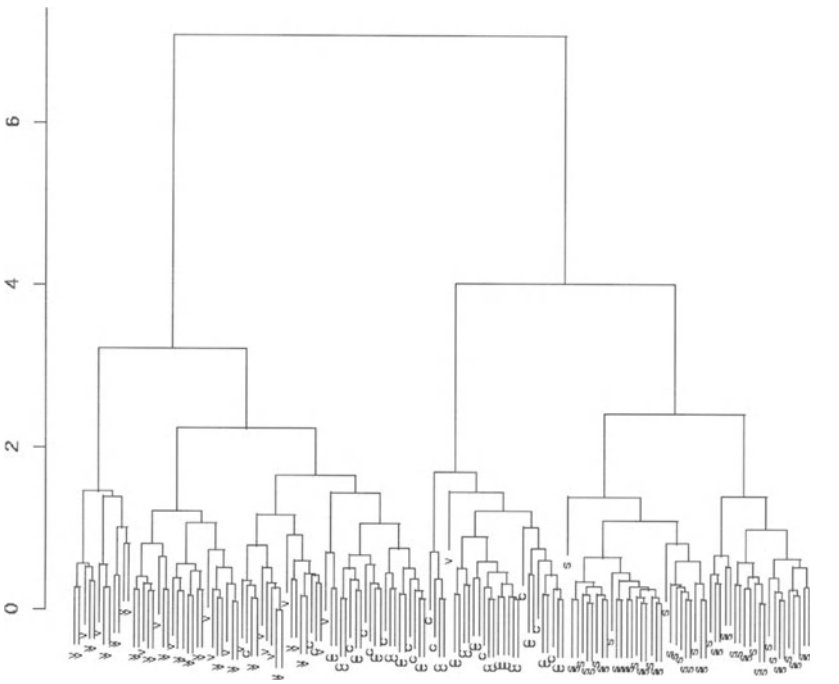

Fig. 4.7 An illustration of hierarchical clustering using the Fisher iris data (Fisher [1936] and Anderson [1935]). The three species of iris - Setosa, Versicolor and Virginica - are denoted S, C and V respectively.

being binning the data (if desired) and selecting the mapping from the observation domain to the color or gray scale of the image. With data images, it is customary to group the data, using a (usually hierarchical) clustering algorithm, so that similar observations lie close to each other. One can also sort the variables, to improve the visual impact of the display.

The first example of a data image that I am familiar with is in Ling [1973]. This paper is worth looking at for the graphics alone (remember that this was written in 1973). The author uses character graphics to display a data image. One version of the data image is described in Minnotte and West [1998].

Shoch and Hupp [1990] use data images to display the progress of a worm, plotting the source IP against the destination IP for traffic during its propagation. This is a simple but effective graphic, showing the worm moving from machine to machine in a staircase pattern in the plot.

The genome analysis community has made good use of the color histogram. For example, Eisen et al. [1998] use a data image to display gene expression as a function of time, with the genes clustered using a hierarchical clustering technique. They depict the dendogram, colored by cluster, next to the color histogram, producing a particularly impressive graphic. These days, with the interest in the Human Genome Project, nearly every issue of the journal Science has a data image in it displaying information about gene expression or such. This is further evidence that the technique is a useful and powerful one.

Figure 4.8 illustrates a color histogram for 300 observations from the 20-dimensional density

$$f(x) = \frac{1}{3}N(0, I) + \frac{1}{3}N(\mu_1, I) + \frac{1}{3}N(\mu_2, I), \tag{4.1}$$

where μ_1 is zero for ten variables and 3 for ten (randomly selected and unknown to the data analyst) variables. Similarly, μ_2 has ten zeros and -3 in ten randomly selected variables. I corresponds to the 20-dimensional identity matrix. (For the purposes of demonstration, there were actually 100 observations generated from each of the components rather than generating data from the distribution of Equation (4.1) directly. The data were then randomized.)

Note that although we call this a "color" histogram, all our graphics are in gray. This is in part due to the desire to reduce the expense of producing (and hence purchasing) this book and in part due to the fact that unless care is taken, color can create unnecessary confusion. The exception is when color can be used to encode specific information, such as a priori groupings of the data. This is used to good effect by the gene expression researchers (Eisen et al. [1998]). It is unnecessary for the data we are interested in here.

Looking at Figure 4.8, it is clear that there are approximately ten variables (rows) that are darker than the others, indicating the effect of the two nonzero means. However, we have no way of telling from this figure whether there are one, two, three, or more populations in the data. All we can really tell is that the variables are not identically distributed.

Figure 4.9 shows the same data as a data image, where both the observations and variables have been grouped (independently) using a complete linkage agglomerative hierarchical clustering algorithm. The three clusters are clearly evident.

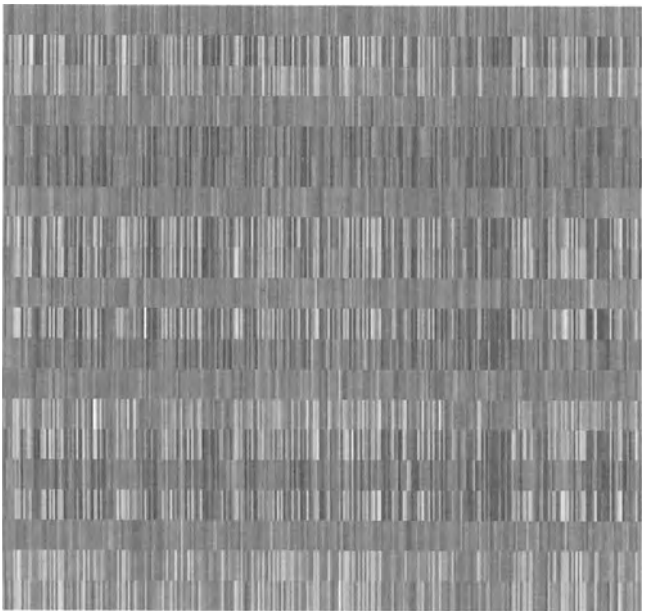

Fig. 4.8 An example of a color histogram for 300 observations of 20-dimensional data from a three component distribution. The x-axis corresponds to the observations and the y-axis corresponds to the variates.

Another way to analyze these data is to compute the interpoint distance matrix and display this as a data image. The interpoint distance matrix is a matrix where the (i, j)th entry is the distance between the ith and jth observations. This is grouped, again with a hierarchical clustering algorithm, and displayed in Figure 4.10. Since the interpoint distance matrix is symmetrical, both the rows and the columns are grouped using the same scheme. This results in a symmetrical image. Black corresponds to small distances (note the diagonal black line, corresponding to a distance of zero between observations and themselves). The three clusters are clearly evident as dark squares along the diagonal.

Figure 4.11 depicts the dendogram associated with the data image of the interpoint distance matrix in Figure 4.10. The three clusters are clearly evident in this figure. In some sense, the data image is a picture of the dendogram, with the added information of the relative values of the observations coded as a gray scale or color value.

One issue that I have not seen addressed is that, like the pairs plots of Section 2.5.2, the data image of the interpoint distance matrix is redundant. It would be interesting to consider informative uses for the upper triangle of the plot. One

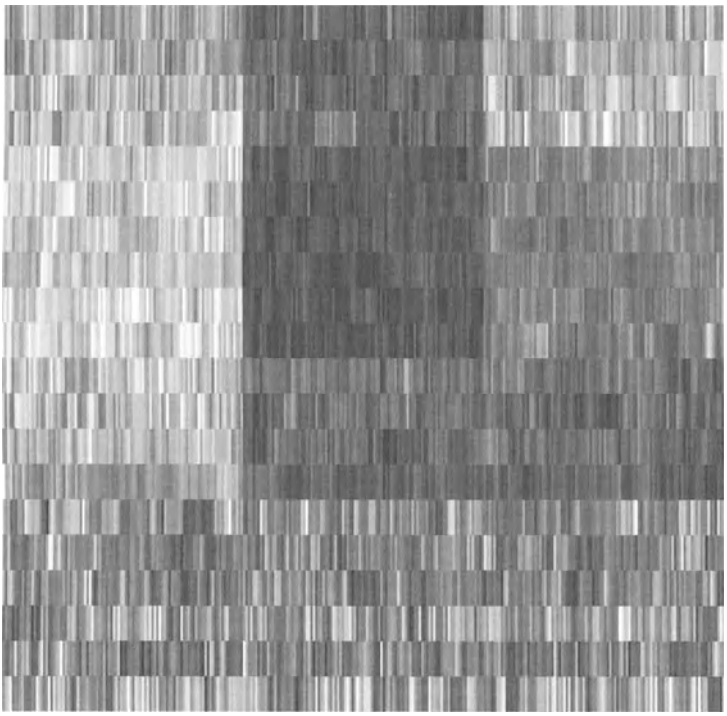

Fig. 4.9 A data image of the data in Figure 4.8. Both the observations and the variables have been grouped with complete linkage hierarchical clustering algorithms.

potential use would be to plot a second distance metric. This is an area for future work.

A use of the data image that to my knowledge has not appeared in the literature is its use to detect outliers. Figure 4.12 depicts the interpoint distance matrix of the data in Figure 4.10, except that three of the observations have been modified to be outliers (by adding 5 to each of their variates). This image has been inverted for display purposes; white now corresponds to small values and black to large, instead of the other way around. Since the diagonal must always be zero in the interpoint distance matrix, one can easily tell the color scheme by examining the diagonal of the image.

The outliers show up clearly in this image as a "v" of dark color in the bottom left of the image. Thus, outliers can be detected visually as either a "v" or a "+" in the data image. Further, this is particularly useful for detecting outlying clusters since these outliers will show up as broader bands of black with a small gray or white square in the center of the "+" (or the vertex of the "v").

This latter is particularly important for computer security. After all, attacks on a computer or network are, by definition, outliers. Provided we can choose an appropriate way to map network data into vectors, we can use data images to

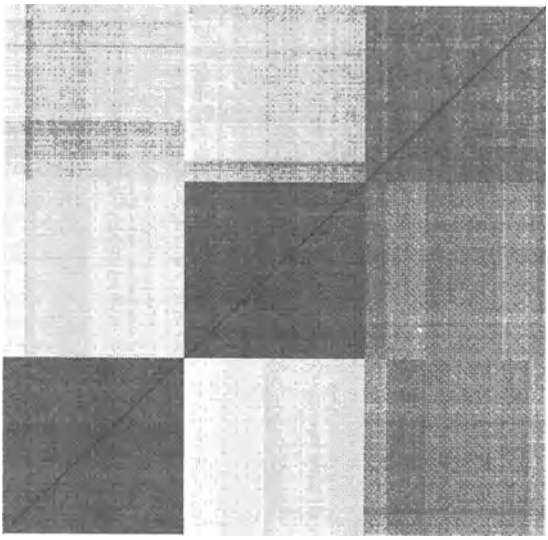

Fig. 4.10 A data image of the interpoint distances of the data depicted in Figures 4.8 and 4.9.

detect these outliers and hence the attacks. This is not a trivial caveat, but it does provide promise for future work.

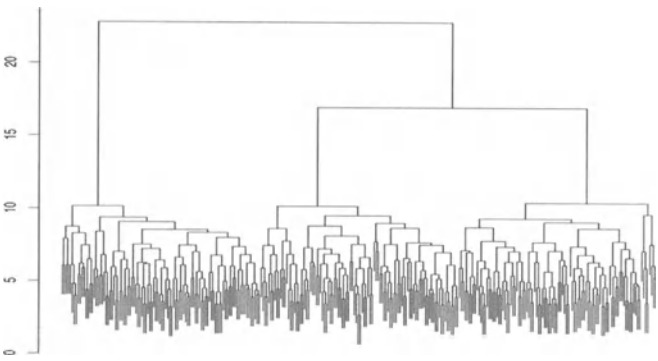

Fig. 4.11 A dendogram of the data depicted in Figures 4.8 through 4.10.

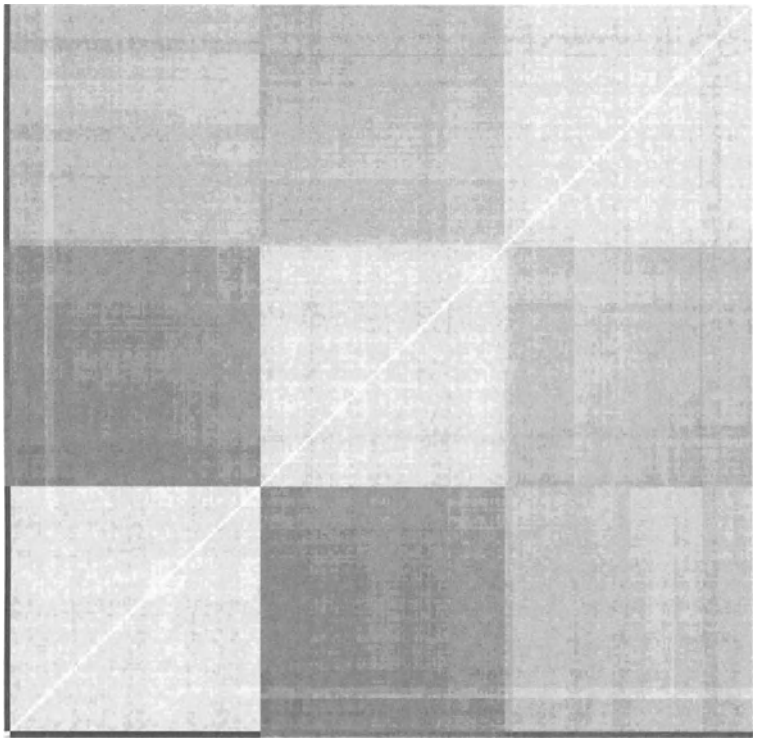

Fig. 4.12 A data image of the interpoint distances of the data depicted in Figures 4.8 where three of the observations have been changed to be outliers. These are clearly evidenced as a dark "v" with vertex in the lower left. The gray scale has been inverted in this image for display purposes.

4.5.3 An Example

An organization wanted to know what kinds of machines (e.g., mail servers, Web servers) were active on its network. This might seem like a strange request, since many organizations have some kind of accreditation procedure that must be followed before a machine can be installed on the network and so have a list of the active machines. Several things can go wrong with this, however:

- Machines can have new applications installed without informing management.
- Machines can be installed without notification of management by simply taking the IP address of a machine that was accredited but is no longer used.
- Some facilities have subnets that are not the purview of the security management. These subnets sit behind a firewall, and the machines on the subnet are not accredited by the normal means. Only the firewall is accredited. The organization that owns the subnet is then responsible for its security and management.

- Some sites (particularly military sites) have several organizations behind their perimeter firewalls. The security managers of the site may not have specific information on the machines used by other organizations.
- Some sites (particularly universities) have very lax accreditation policies and allow quite a bit more freedom on their networks.

The organization that commissioned the study does not want its data used, even after scrambling the IP addresses. Instead, we collected data from another network to use as an example of the approach.

To investigate the activity on the network, we collected data for slightly less than 2 months. These data consisted of hourly counts, where for each IP address/port pair we counted the number of outgoing SYN/ACK packets within an hour. The network was a Class B network, consisting of all machines with a 10.10.x.x IP address (the IP addresses have been changed throughout). The active machines are displayed in Figure 4.13. In this figure, the axes correspond to the third and fourth octets of the IP address, and a black dot corresponds to a machine with activity during the two-month period. Thus, the coordinate (23,192) corresponds to the machine 10.10.23.192. In order to further protect the information about the network, the IP addresses have been scrambled.

There are 64,270 machines "active" on this network (Figure 4.13, upper left). If we consider only those machines with four or more packets (Figure 4.13, lower right), there are only 929 machines. What is going on? Are there really 64K machines on this network? A DNS lookup at the site gives an answer much closer to 2K registered machines. The answer, I believe, is a helpful firewall. My guess is that the site has a proxying firewall, which initiates the session for the protected network prior to checking to see whether the destination machine exists and is accepting connections. Whether this is intended or not is unclear. The site has, off and on, had a proxying firewall but has not been consistent in its use. I have not been able to verify that this is what is happening in these data. I do know that several large scans appear in the data, which would account for access attempts at nearly the entire address space. This would not normally result in outgoing SYN/ACK packets from nonexistent machines, however, so something must be producing these packets.

To get an idea of the amount of data in this two-month period, Figure 4.14 shows the (natural) log of the number of outgoing SYN/ACK packets to the 929 machines indicated by Figure 4.13 (bottom right), sorted by activity. Note that the most active machines have approximately 3 million connections in this time period, which corresponds to 50,000 connections per day. Some of this is a result of scans, some due to services such as Web and FTP, which generate many connections per session, and some due to hosting multiple services on one machine.

The same visualization idea as in Figure 4.13 can be applied to the ports accessed. By treating the 2-byte port number as two individual coordinates between 0 and 255, we can plot the port accesses in an image in much the way we did with the IP addresses. Figure 4.15 shows the port accesses in the data. There are 19,621 distinct ports accessed (Figure 4.15, upper image). If we restrict to only those ports accessed at least four times, this reduces to 56 distinct ports (Figure 4.15, lower image). These correspond to the "active" applications on the network.

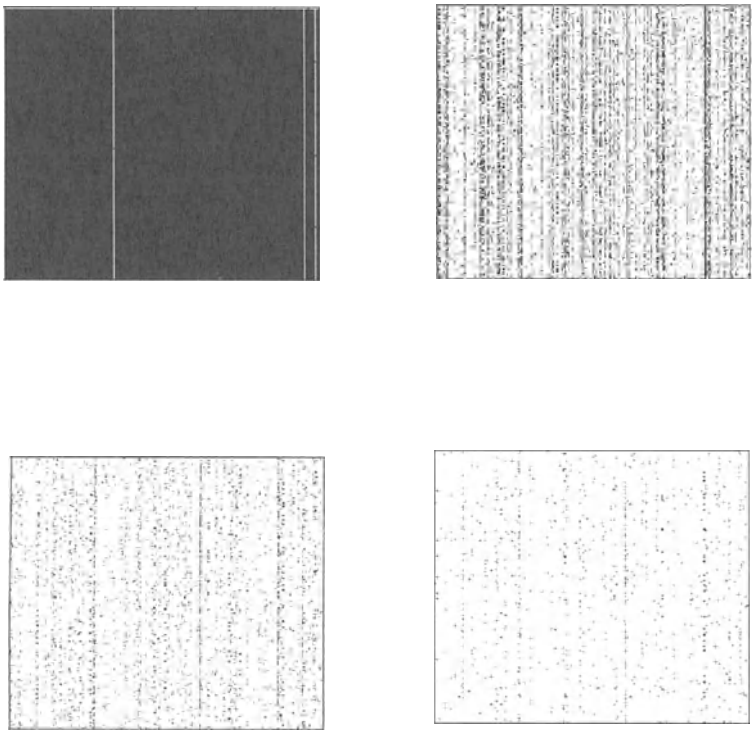

Fig. 4.13 Active machines on a Class B network. The axes correspond to the last two octets. The upper left image shows all machines (in black) that have had at least one outgoing SYN/ACK packet in a two-month period. The upper right image consists of those machines with at least two outgoing SYN/ACK packets. The lower left and right images are those with three or four outgoing SYN/ACK packets, respectively.

In both examples, the number four is chosen arbitrarily. We want to eliminate nonexistent machines, and ports that are used very rarely, because we are trying to obtain an understanding of "normal" behavior. If we were looking for "abnormal" behavior, we might take a different approach.

The first few active ports (ordered by port number, not by amount of activity) are:

21 FTP

22 SSH (secure shell)

23 Telnet

25 SMTP (email)

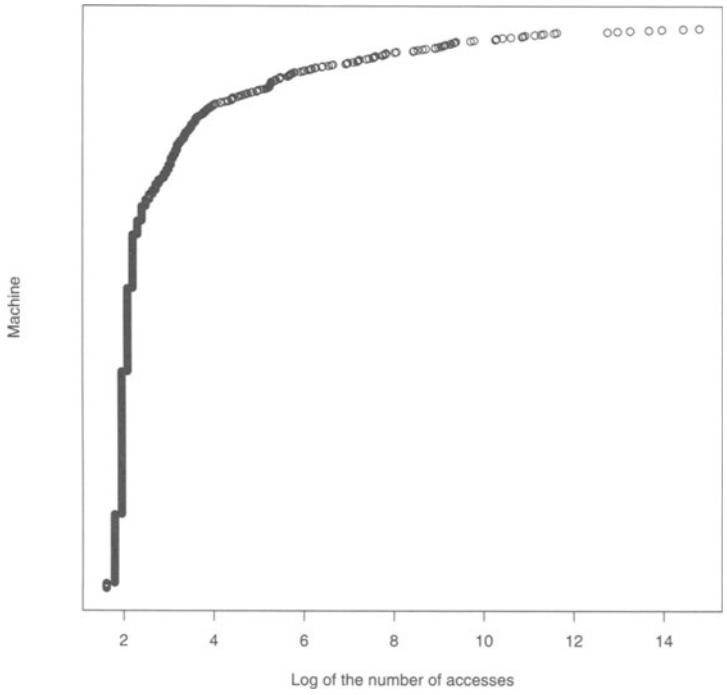

Fig. 4.14 Dot plot of the log of the number of accesses to the 929 machines in Figure 4.13, bottom right.

53 DNS (domain name service)

79 Finger

80 HTTP (WWW)

These applications will probably be found (with the possible exception of finger) on nearly any network in the world.

It is interesting to consider the upper image in Figure 4.15. There is a band of active ports in the low number ports, which makes sense. These are the common applications such as those listed earlier. They are also applications that would be scanned for by an attacker. The band near the middle corresponds to the large number of applications that allocate a port for data transfers. These port numbers don't necessarily mean anything by themselves, because any application can use them. They have a low number of accesses, as evidenced by the lower image in Figure 4.15. This is primarily a result of the fact that there are a large number of ports from which to choose, so most machines do not recycle them very quickly.

The 56 ports with at least four accesses are depicted in Figure 4.16. This is a dot plot of the log of the number of accesses. It is not surprising that email is the

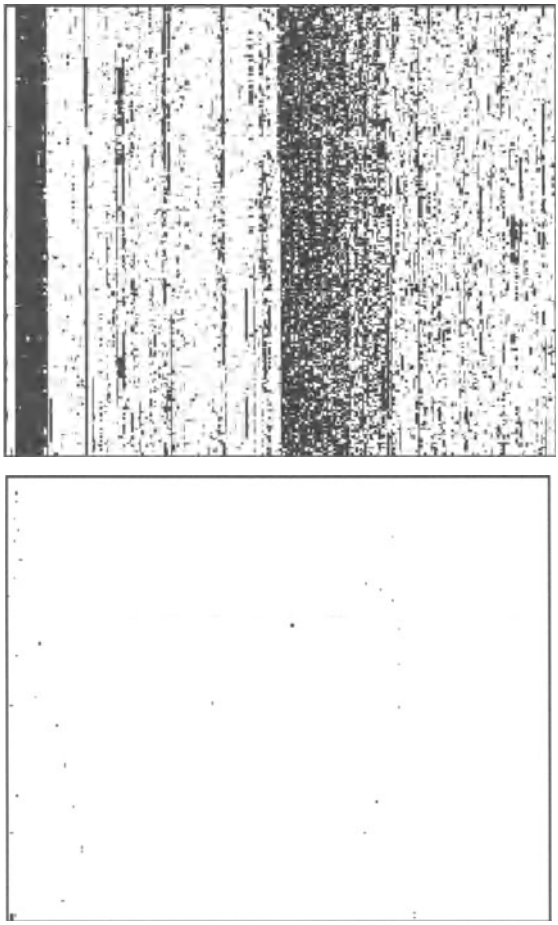

Fig. 4.15 Active ports for the machines in Figure 4.13. The upper image corresponds to all active ports, while the lower image corresponds to those ports for which some machine responded at least four distinct times. The axes correspond to the first and second bytes of the port number.

most popular port by far. Secure shell and secure Web (https) are the next most common, which is also reasonable for a relatively security-conscious site.

Figure 4.17 depicts the color histogram of the activity vectors. In this plot it is difficult to discern much structure. It is clear that there are two or three ports that are quite common across many machines. Thus, we would feel confident stating that there appear to be several clusters (those that have these services and those that do not), but it is impossible to determine the cluster structure from this unsorted plot.

Figure 4.18 depicts the data image of the data from Figure 4.17. Now, the clusters are much clearer. There are a number of small clusters, which are somewhat difficult to discern, followed by four or five clear clusters corresponding to the use of four distinct ports. Note that even in this picture it is difficult to decide exactly

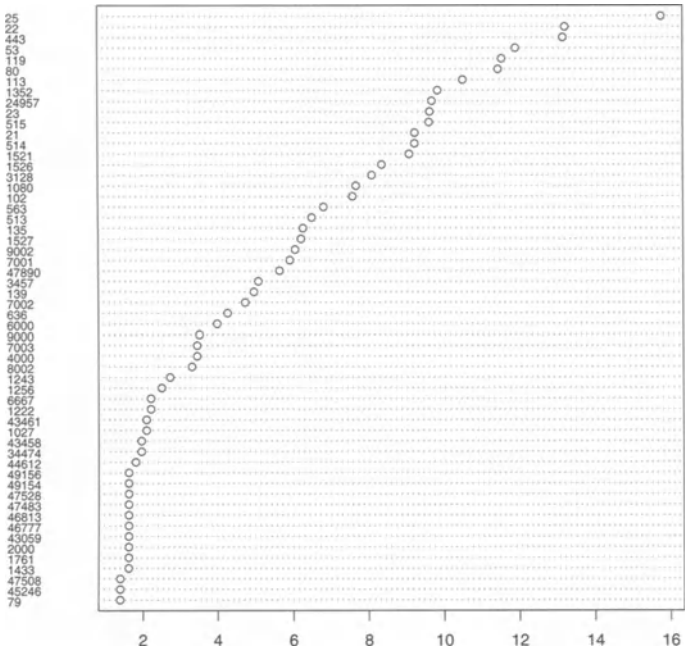

Fig. 4.16 Dot plot of the log of the number of accesses to the 56 ports depicted in the lower image in Figure 4.15. The vertical axis corresponds to port numbers, while the horizontal axis corresponds to the log of the number of accesses.

how many distinct clusters there are in these data. This is pretty much always the case when dealing with real data.

The dendogram for the activity vectors is depicted in Figure 4.19. This is difficult to interpret due to overplotting, which results from the large number of observations. This figure illustrates the difficulty of adequately displaying large dendograms within the constraints of static media (such as paper).

Figure 4.19 also points to a subtle problem with visualization techniques such as those discussed in this section. When the number of observations is large, there is a very real probability of overplotting. In fact, with color histograms, overplotting is certain whenever the number of observations exceeds the number of pixels in the image. This is precisely why binning is implemented (hence the name "color histogram"). Thus, in Figure 4.18, we can really see only a portion of the observations and hence the clusters.

To illustrate this, consider Figure 4.20. Here we have zoomed in to the right-hand side of Figure 4.18, the last 150 machines. We can see several things in this plot. For example, we note that there are several singleton machines on the far right that did not show up in the original plot. Further, we see some structure that

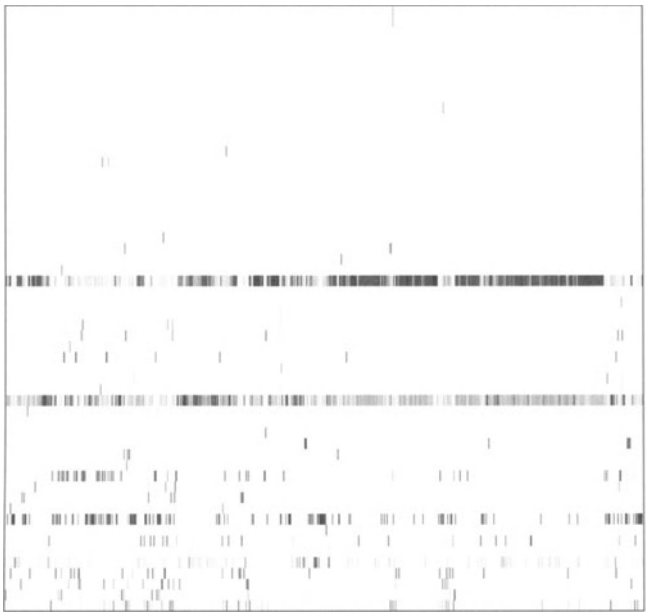

Fig. 4.17 The activity vectors of the 929 machines of Figure 4.13. The machines correspond to the x-axis, while the port numbers correspond to the y-axis.

was not apparent. This is due in part to the overplotting and in part to the fact that the zoomed-in region has a smaller dynamic range, which allows the use of more gray values for these regions in the image.

As mentioned earlier, it is sometimes easier to see the clusters if one uses the interpoint distance matrix instead of the raw data. This is particularly true if the data are very high-dimensional. For example, if we were to keep all the ports in the activity vectors, we would be unable to view the color histogram easily (unless we happened to have 65 screens attached to our computer, an unlikely event). Figure 4.21 depicts the color histogram for the interpoint distance matrix. The structure is quite apparent in this image.

4.5.4 Statistical Anomaly Detection

As discussed earlier, we are interested in collecting statistics on the activity on the network with a view toward detecting anomalies in the traffic on the network. One approach is described later, the NIDES system, described in Section 5.3. NIDES was really designed to be a host-based intrusion detection system, but the basic idea is easily adapted to apply to network monitoring, and in fact it is the statistical

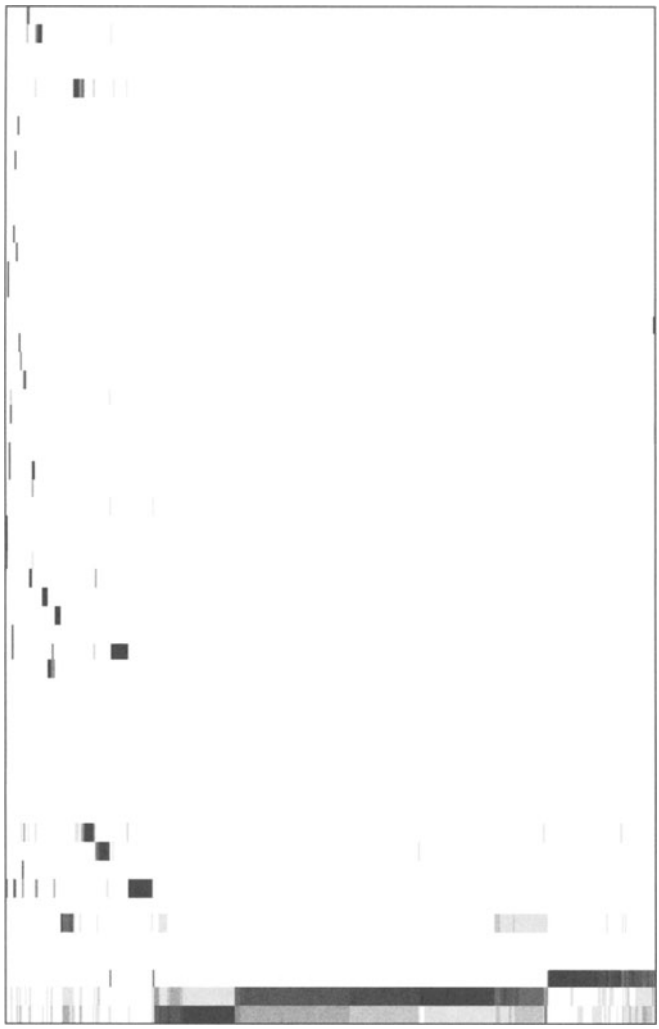

Fig. 4.18 Data image of the data of Figure 4.17. Again, the x-axis corresponds to machines, while the y-axis corresponds to port numbers. The port numbers have been resorted using a hierarchical clustering technique to improve the visual impact of the plot.

engine in EMERALD (Section 4.6). In this section, we will look at a simpler implementation, described in Marchette [1999].

The idea, as before, is to collect activity vectors for each machine. These activity vectors are defined to be the proportion of accesses (TCP SYN packets or UDP packets) to a given port on a given machine over a given time period. Thus,

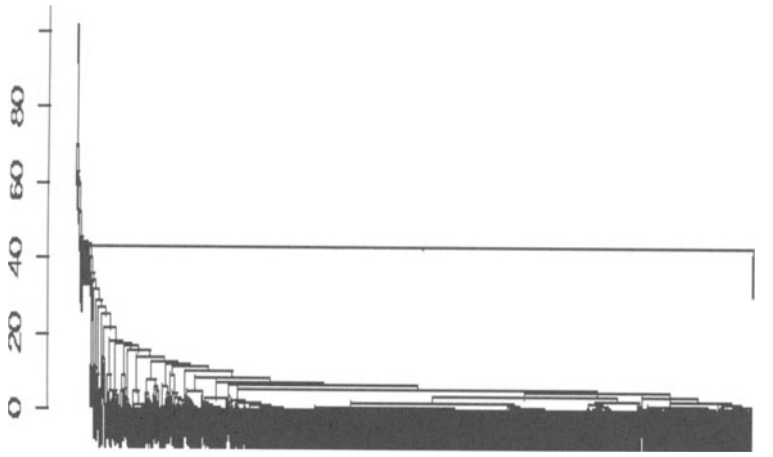

Fig. 4.19 A dendogram for the data plotted in Figure 4.18.

an activity vector for a machine consists of 2*65,536 values (corresponding to the TCP and UDP ports available).

This is a very high-dimensional vector, and some domain knowledge is required. Recall that while the first 1024 or so ports are generally assigned to specific services, the higher ports are often assigned by other services at run time. For example, FTP will assign ports (usually in the low thousands) for data transfers, but which ports get used is implementation- and machine-dependent. Thus, for ports above 1024, one is really interested in ranges of ports more than individual ports. In Marchette [1999], this approach was taken to its extreme. Ports 0–1024 were treated individually, while all ports above 1024 were considered "big ports" and grouped together. Thus, the vectors are reduced to 2*(1024+1)-dimensional.

Alternatively, one could define several ranges in the "big port" range, such as ports 6000–6063 for X window access, FTP data-transfer port ranges, traceroute port ranges, and so on. Some of the "big ports," for example 2049, which is used by NFS, should probably be treated individually in the same manner as the low-numbered ports. Even within the low-numbered ports, one could group the ports corresponding to related services or ports corresponding to services not offered on the network.

For each machine, an activity vector is defined, consisting of the proportion of accesses to the ports or port ranges. We view these proportions as estimates of the probability of accessing the given port. The idea behind using activity vectors for anomaly detection is to flag as anomalous any port access that has a low probability of access.

In the simplest form, these activity vectors can be a method for constructing individual tcpdump filters. If we set a threshold on the activity values, every port access for those ports above the threshold is "normal" and should thus be ignored.

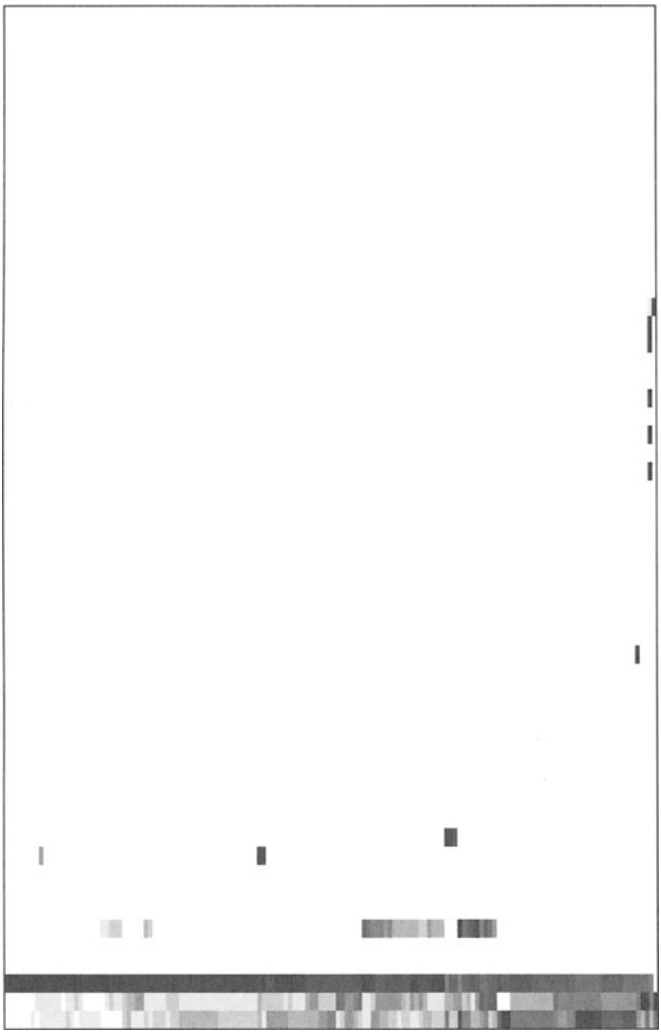

Fig. 4.20 The last 150 observations in Figure 4.18, zoomed in for improved resolution.

A filter that itemizes these can be used to provide a "personal SHADOW" system for the machine.

For example, suppose for machine 10.10.1.23 the ports above threshold consist of ports 22, 23, 514 TCP, and 2049 UDP. Place the following filter in the file "myfilter":

```
dst host 10.10.1.23 and

    not (
```

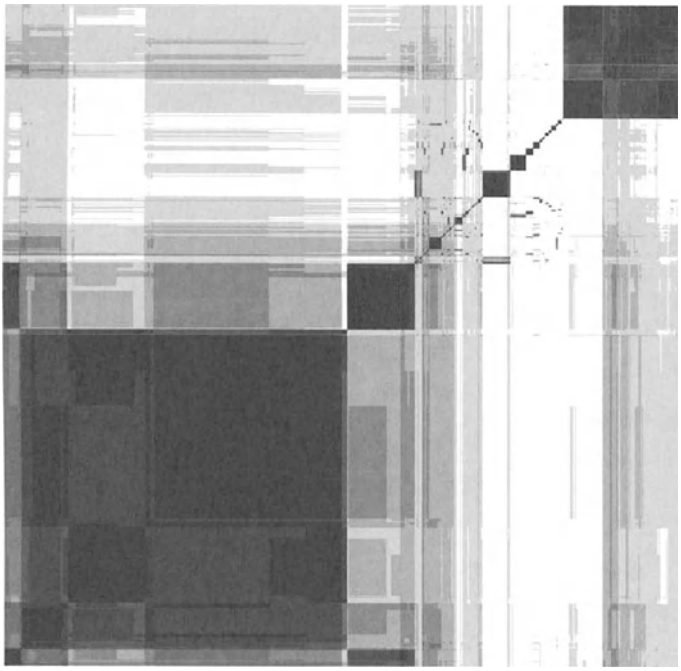

Fig. 4.21 Color histogram for the interpoint distance matrix for the data in Figure 4.18.

```
        (tcp and (dst port 22 or dst port 23 or dst port 514))
        or
        (udp and dst port 2049)
)
```

Running

```
tcpdump -F myfilter | logger
```

will result in any accesses that are "abnormal" being logged to the syslog file. Like the SHADOW system described in Section 4.4, this will alert the system administrator when undesirable network activity is aimed at the system.

This simple filter is inadequate for most real systems because of services such as Web and FTP, which use many ports. These can be handled, however, by making the filter a little more complicated. I will leave this as an exercise to the reader.

Note that the preceding command must be run as root and that it puts the network interface in promiscuous mode. In some organizations, a network card in promiscuous mode is considered a threat (after all, it copies all the packets that pass it, whether destined for that host or not), so it is probably a good idea to use the "-p" flag of tcpdump to keep it from going promiscuous. It should be further noted that even though you will no longer be seeing packets to other hosts, you

Table 4.10 Attacks identified in the data used in Marchette [1999] to demonstrate activity profiling.

Attack Type	Number
Bad Ports (111, 161, etc)	5
Suspicious Telnets	6
Suspicious FTPs	1
Netbios Probs	6
Zone Transfers (53 TCP)	2
Port Scans	1
Traceroute	1
Finger Probe	1
NNTP	1
NFS	1
Miscellaneous Ports	2

will be seeing packets belonging to other users on your system. For this reason, care should be taken to ensure that their privacy is not invaded. Even this level of monitoring may be against the policy of your organization or even the laws of your country.

Once we have constructed the activity profiles for all the machines on the network, we need to use these to detect abnormal access attempts. A set of experiments is described in Marchette [1999]. Data were collected for a network consisting of 993 active machines. Activity vectors were computed for the machines using the data from a single month. A second month's worth of data was used to determine the performance of the approach in detecting attacks. These data consisted of a total of approximately 1.7 million TCP SYN and UDP packets in each month. There were 27 attacks identified within the data. The task was to determine the number of attacks detected (designated "abnormal") at different threshold values and therefore produce an ROC curve. (See Section 3.3 for a discussion of ROC curves.)

The data consist exclusively of incoming SYN packets and UDP packets from outside the network. There was no attempt to determine whether the machines had responded to the connection attempts (in fact, this information was lost due to the decision to retain only incoming TCP SYN and UDP packets). The attacks were detected by an experienced analyst investigating the reports from a SHADOW system. The attacks were broken into 11 groups, based on the type of attack. The groups are listed in Table 4.10.

The port scan was an unusual one, which looked for services running on ports above 1024. This made it potentially difficult to detect using the method described in Marchette [1999] because of the grouping of all high-port accesses into a single bin. Similarly, the traceroute attack, which also shows up in the high ports, would be impossible to detect on a machine that normally had accesses on high ports.

A note on the traceroute attack is in order here. A reasonable person might say that traceroute is not an attack. It is, after all, a common utility that has many

legitimate uses. It may even seem difficult at first to identify some illegitimate uses. There are two that come to mind, however. Obviously, since traceroute provides information about the routers between two machines, it can provide information about a network's internal routers. A map of the internal routers can provide an attacker with very useful information.

There is another, more sinister reason to be suspicious of traceroute access to a network, however. Imagine that an attacker wanted to completely disable your network connection. Imagine further that you are a very well-protected site, say a military site, and a direct attack against your site may not be desirable. How would the attacker accomplish this goal? One way would be to attack not your site but the ISP(s) that service your site. By taking out the organization(s) that provide your connectivity, an attacker can completely remove your site from the Internet.

What this reduces to is that somehow the attacker needs to find out what routers service your site. This is where traceroute comes in. By running traceroutes to machines at your site from a variety of sources, the attacker can (in principle) determine all the (active) routers connecting your site to the Internet. By attacking these routers (which you have no power to protect because they are the property of another organization), your adversary can damage or eliminate your ability to function.

Of course, as we have seen in Section 2.4, there are people using traceroute to perform network mappings all the time for perfectly legitimate purposes. Because of the accessibility of these data, it may very well be that the traceroute "attack" is now unnecessary because the desired information is freely available.

Whether the particular traceroute in the data for this experiment was an attack is a matter of speculation. There were other factors that made it suspicious, so it was left in the data.

Since these are real data, there is no guarantee that the attacks detected are the only ones that occur in the data. These data were collected early on in the SHADOW project, so it is reasonable to assume that some attacks were missed. Also, some of the "suspicious telnets" were identified because of their source IP address (for example, coming from a foreign country), information that is not retained in the activity vectors.

Using the activity profiles for a machine, new packets can be scored as to their "normality" by considering the probability that a packet of that type would be seen coming to the destination machine. Low-probability packets are flagged as being suspicious.

In addition to looking at individual machines, Marchette [1999] looked at clustering machines into groups with similar activity profiles, then used an average activity level for the cluster profiles. Two different clustering techniques were used and compared to an approach using the individual machine activity profiles.

The first method used is the k-means algorithm, a standard clustering technique (see Everitt [1993]). The number of clusters, k, is assumed known. The algorithm is then as follows:

The k-means Clustering Algorithm

1. Initialize the k cluster centers (for example at k randomly chosen observations).

2. While the centers change do
   (a) Assign the data to the cluster with the closest center.
   (b) Recompute the centers as the mean (or median or other measure of location) of the data in the cluster.
3. Return the clusters.

The k-means algorithm is extremely easy to implement and works well when the clusters are well-separated and spherical. There are a number of issues that need to be addressed in applying it, however. First, the number of clusters needs to be known a priori. The selection of distance metric is another important issue. Finally, the choice of center definition (mean, median, or some other appropriate statistic) needs to be made.

In general, it is a difficult task to decide how many groups there are in a data set. Usually, one either uses domain knowledge for this or some exploratory data analysis and visualization, which, as we have seen, can be difficult for very high-dimensional data. It should be noted that the k-means algorithm can produce strange answers if the number of clusters chosen does not match the data. For example, running the k-means algorithm on normal data with $k = 2$ often results in splitting the single cluster in half. Worse, rerunning with a different center will often result in a different split (for example, east-to-west one time, north-to-south another), so, like all clustering techniques, the clusters should not be taken at face value without some analysis as to their appropriateness to the data. Interested readers should consult one of the many good books on clustering, such as Everitt [1993].

Although the distance metric chosen can be critical to the performance of any clustering algorithm, most practitioners choose either Euclidean distance (l_2) or absolute distance (l_1) unless something about the problem domain suggests a metric. In Marchette [1999], the Euclidean distance was used throughout. Similarly, the mean is usually used for the center, and this was the case in Marchette [1999].

A clustering using an (arbitrary) value of 10 for k is shown in Figure 4.22. These are small color histograms, one for each cluster. The vertical axis corresponds to machine, while the horizontal corresponds to port number. Only those ports for which at least one machine had a probability above 0.2 of activity are shown. One of the clusters contains only a single observation and hence is not shown. The fact that a cluster contains a single observation is evidence that the choice of 10 for k was not appropriate or that the machine in question was an outlier.

One thing that is immediately noticeable about the images in Figure 4.22 is that the clusters are determined primarily by one to three ports. Thus, the activity on a machine can be determined in large part by the ports that have the most activity, and these are typically a small number of ports. This is intuitively reasonable, given that most (properly configured) machines provide a small number of services to outsiders. However, at least one of the clusters (cluster number four, counting from the top left) shows quite a bit of variability in the observations. This probably consists primarily of outliers to other clusters.

A drawback to this kind of display is that it is difficult to determine the port numbers associated with the clusters. This information is of course available and

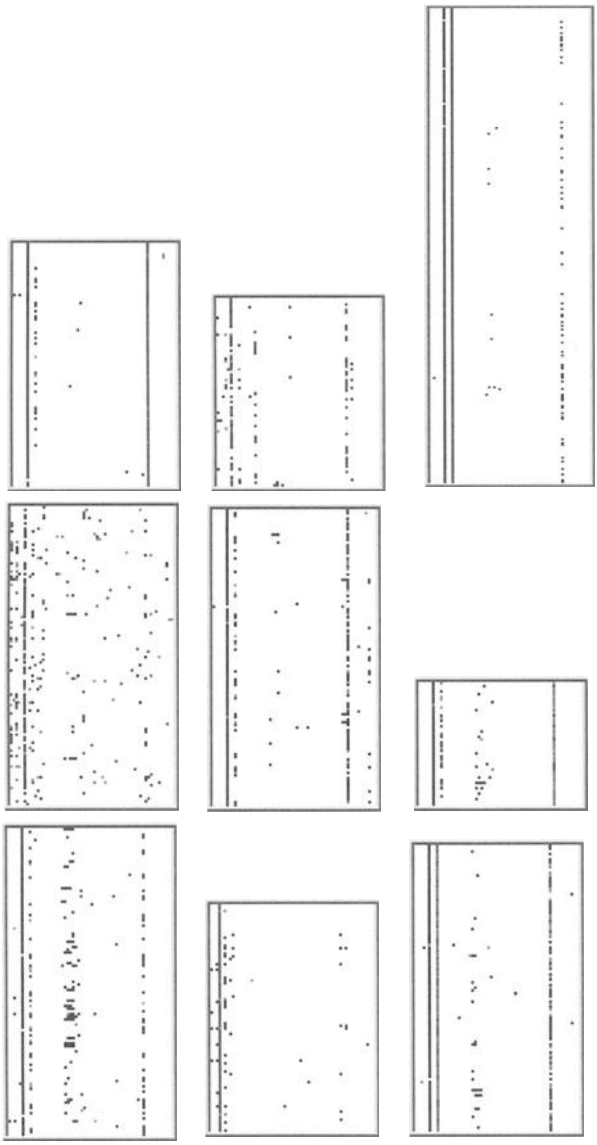

Fig. 4.22 Clusters from the k-means clustering of 993 machines. Each cluster is depicted as a separate data image. The x-axis corresponds to port number whereas the y-axis corresponds to the machine.

can be used to characterize the clusters further. We will see this later when we consider the problem of determining the types of activity available on a network.

The second clustering technique used in Marchette [1999] is the approximate distance clustering (ADC) technique described in Cowen and Priebe [1997a] and Cowen and Priebe [1997b]. The idea is to select out a subset of the data, referred to as the witness set, which acts as a kind of prototype for the data. For each

observation, the smallest distance to any element of the witness set is computed. The observation is then projected to one-dimensional data by taking this distance as the value of the projected observation. The method can be extended to utilize several witness sets, in which case the projecting dimension is the number of witness sets.

Therefore, given a set of observations X and a witness set W, the ADC projection is

$$d(x) = \arg\min_{w \in W} d(x, w). \tag{4.2}$$

The ADC approach taken in Marchette [1999] is to project the data to the real line using Equation (4.2) and then cluster the one-dimensional data. The clustering on the one-dimensional data was performed by modeling the data as a mixture of normals. See McLachlan and Basford [1988], Titterington et al. [1985], or McLachlan and Krishnan [1997] for more information on mixture models in general. The equation for a (univariate) normal mixture density is

$$f(x) = \sum_{j=1}^{m} \pi_j \phi(x; \mu_j, \sigma_j^2) \tag{4.3}$$

where the π's are positive and sum to one, ϕ is the normal density, μ_j is the mean, and σ_j^2 is the variance.

As with the k-means algorithm, the number of clusters must be chosen. This can be done a priori as in the k-means algorithm described above, or the number of terms can be estimated from the data.

The EM algorithm (see page 50; also McLachlan and Krishnan [1997]) for the normal mixture parameters produces the following update equations, given data $x_1, ..., x_n$:

$$\tau_{ij}^{t+1} = \frac{\pi_j^t \phi(x_i; \mu_j^t, v_j^t)}{f^t(x_i)}, \tag{4.4}$$

$$\mu_j^{t+1} = \sum_{i=1}^{n} \tau_{ij}^t x_i, \tag{4.5}$$

$$v_j^{t+1} = \sum_{i=1}^{n} \tau_{ij}^t (x_i - \mu_j^t)^2, \tag{4.6}$$

$$\pi_j^{t+1} = \sum_{i=1}^{n} \tau_{ij}^t, \tag{4.7}$$

where $v_j = \sigma_j^2$ and the superscript t indicates the iteration number. There is a similar formula for multivariate mixtures. See McLachlan and Basford [1988], Titterington et al. [1985], or McLachlan and Peel [2000] for more details. The procedure is to start with an initial guess at the parameters, then run the iterations defined in Equations (4.4)–(4.7) until a convergence criterion is met (usually until the change in the log likelihood is small).

In order to estimate the number of terms in the mixture model, the alternating kernel and mixture density estimator (AKMDE) of Priebe and Marchette [2000]

was used. As with most methods for determining the number of components of a mixture, this operates by starting with a single component, and then testing to decide whether a second component is warranted. This continues until the test fails to support a new component. In order to do the test, the AKMDE compares the mixture model to a kernel estimator constructed using the mixture model. If the kernel estimator exhibits structure not accounted for by the mixture model, then a new term is added.

The resulting model is then a mixture of normals

$$f(x) = \sum_{j=1}^{\hat{m}} \pi_j \phi(x, \mu_j, \sigma_j^2), \tag{4.8}$$

where the π's are the mixing coefficients and ϕ is the normal density as before. In this case, the number of terms is estimated from the data, as is indicated by the $\hat{m}$.

The kernel estimator is a commonly used nonparametric density estimate (Silverman [1986]). It is similar to the histogram, but instead of counting the number of observations within a bin, the kernel estimator in effect counts the number of bins at an observation. Specifically, given observations $x_1, ..., x_n$, the kernel estimator is defined as

$$f(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right). \tag{4.9}$$

The kernel K is generally taken to be a symmetric probability density function. We will take it to be the standard normal density $\phi(x, 0, 1)$. The parameter h, called the bandwidth, controls the smoothness of the estimate in much the same way that the bin width controls the quality of a histogram. Large values of h produce very smooth, broad estimates, whereas small values of h produce rough, spiky estimates.

The choice of h is therefore critical to the performance of the estimator. There are many approaches to the selection of h, but this would take us far beyond the scope of this book. See Wand and Jones [1995] for more information on bandwidth selection and kernel estimation in general. One common technique is to use a pilot estimate such as a normal density fit to the data. The bandwidth is then chosen to be the optimal one (in the sense of mean integrated squared error) for the pilot density.

In the AKMDE, the mixture is used as the pilot estimator. Then, the kernel estimator is the optimal one under the assumption that the data are distributed as the mixture density. However, the AKMDE uses a modification of the kernel estimator, the filtered kernel estimator (FKE), that is better suited to the modeling of mixture densities. Consider the density depicted in Figure 4.23. This is the mixture

$$f(x) = 0.9\phi(x, 0, 1) + 0.1\phi(x, 3, 0.01). \tag{4.10}$$

A single bandwidth estimator on data drawn from this distribution would have difficulty obtaining a good estimate. The component on the left requires a relatively large bandwidth, whereas the one on the right requires a smaller bandwidth. For

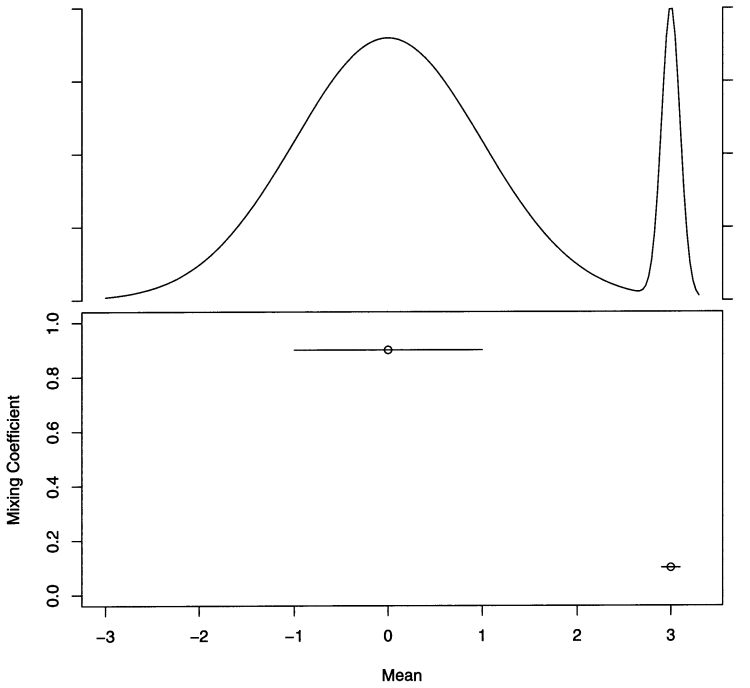

Fig. 4.23 The mixture model of Equation (4.10). The top pane depicts the density and the bottom depicts the mixture model. The y-axis of the bottom pane denotes the mixing proportion of the component. The x-axis denotes the component mean. Each component is plotted as an interval indicating a one-σ range on either side of the mean, which is plotted as a circle.

100 observations, the optimal bandwidths are approximately 0.5 on the left and 0.05 on the right. Figure 4.24 depicts the two kernel estimators with these "optimal" bandwidths. Note that each estimate does a good job on the mode for which the bandwidth is "optimal," and a poor job on the other. Intuitively, the "right" thing to do for this density is to use two bandwidths. This is the idea behind the filtered kernel estimator (FKE) in Marchette [1996], and Marchette et al. [1996].

The filtered kernel estimator provides a multi-bandwidth kernel estimator driven by a pilot normal mixture model. Given a mixture estimate as in Equation (4.8) (referred to as the "filtering mixture"), the FKE is defined to be

$$\begin{aligned} fke(x) &= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{\pi_j\phi(x,\mu_j,\sigma_j^2)}{h\sigma_j f(x)}K\left(\frac{x-x_i}{h\sigma_j}\right) \\ &= \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m}\frac{\rho_j(x)}{h_j}K\left(\frac{x-x_i}{h_j}\right), \end{aligned} \tag{4.11}$$

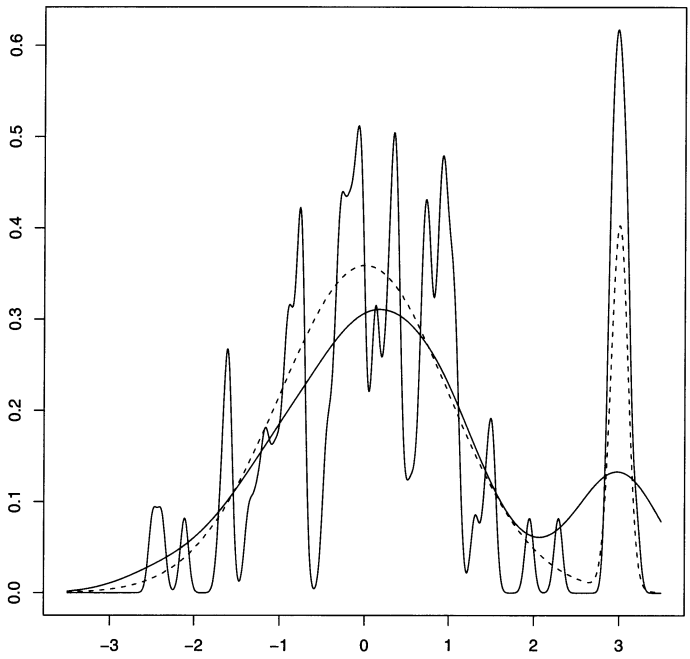

Fig. 4.24 Two kernel estimators of 100 observations drawn from the mixture model of Equation (4.10). The true density is depicted as a dotted curve. The kernel estimators have bandwidths of 0.5 and 0.05, optimal for the two components.

where $h_j = h\sigma_j$. Note that there is still a single bandwidth to be chosen in the formulation of Equation (4.11). However, because of the filtering mixture $f(x)$ and the variances σ_j^2, there are actually m different bandwidths in the estimator. The range of influence of the individual bandwidths is controlled by the posterior probability functions $\rho_j(x)$.

The bandwidth, either for the kernel estimator or the FKE, can be chosen by minimizing the mean integrated squared error (MISE),

$$\text{MISE}(g, \hat{g}) = E\left[\int_{-\infty}^{\infty} (g(x) - \hat{g}(x))^2 dx\right]. \tag{4.12}$$

In either case, the unknown function g is replaced by a pilot estimate. In the case of the FKE, the pilot estimate used is the filtering mixture. Thus, given a pilot estimate in place of g, the bandwidth h is chosen to minimize the MISE.

In practice, the minimization must be done numerically. In order to simplify the calculation, an approximation is made via the expansion of g as a Taylor series. The details can be found in Silverman [1986], Wand and Jones [1995], Marchette [1996], and Marchette et al. [1996]. Suffice it to say that a relatively

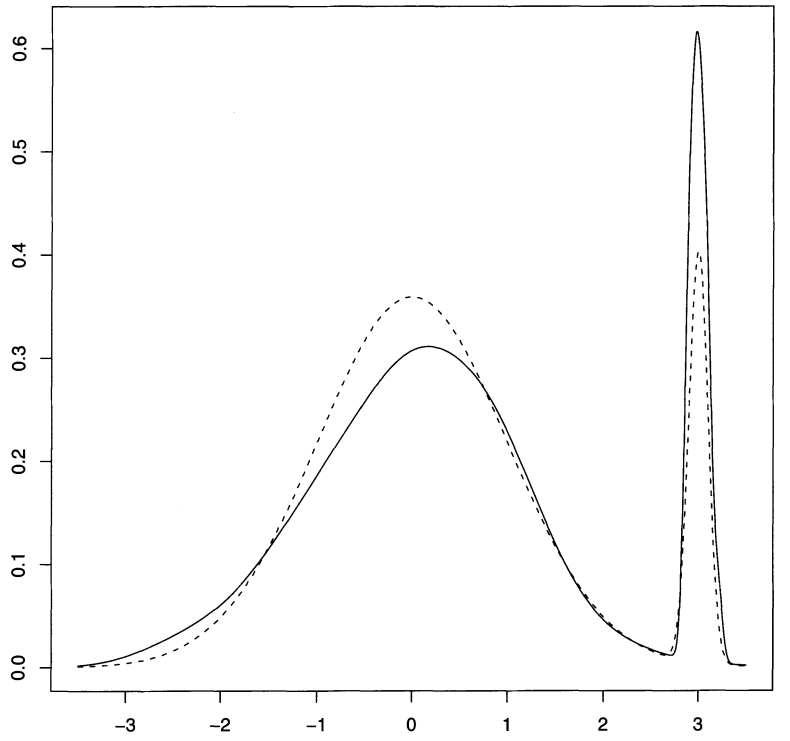

Fig. 4.25 The filtered kernel estimator of the data from Figure 4.9 using the mixture model of Equation (4.10) as the filtering density. The true density is depicted as a dotted curve. The filtered kernel estimator has bandwidths of 0.5 and 0.05, optimal for the two components.

straightforward calculation can provide a reasonably good bandwidth using this technique.

Figure 4.25 depicts the filtered kernel estimator for the same data used in Figure 4.9. Note that in this case the FKE uses essentially the "correct" modes of the two kernel estimators pasted together. This provides an estimator with the correct amount of smoothness in the different regions of the data.

Given two models for a data set, one method for choosing one model over the other is the Akaike information criterion (AIC), Akaike [1974]. This compares the increase in the likelihood of one model over the other, penalized by the increase in the number of parameters. In particular, for the mixture model, the criterion is to reject the new component if $3 - \delta(\text{likelihood}) > 0$ (3 being the number of parameters added to the model by a new term).

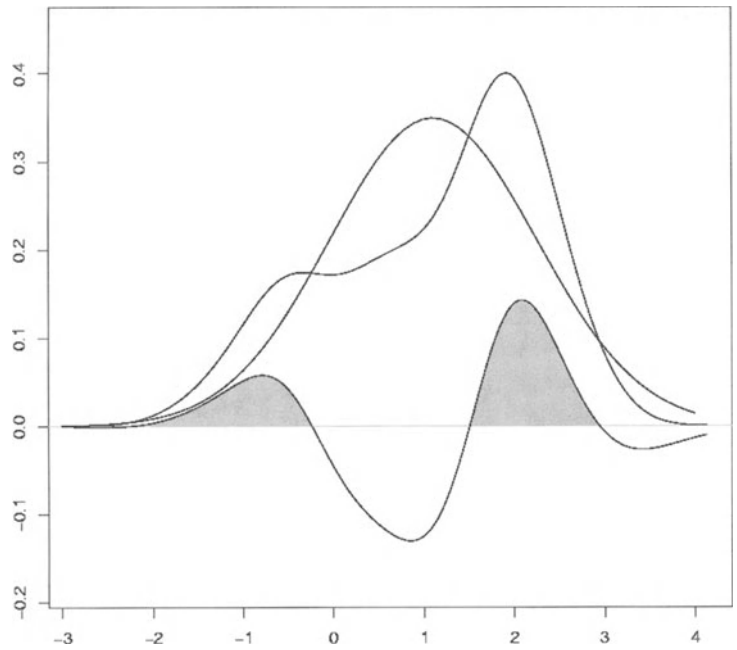

Fig. 4.26 An example of computing the excess mass between a mixture model and a filtered kernel estimator. The mixture is a single normal fit to the data, the filtered kernel estimator using the mixture as the filtering mixture is shown, and the difference curve is shown on the bottom, with the excess mass colored in gray. The true mixture components have means at 0 and 2.

Now, we can describe the AKMDE in more detail. Starting with a mixture of m components, a filtered kernel estimator is formed using the mixture as the filtering mixture. A new $m+1$ component is formed by adding a term corresponding to the maximal excess mass between the FKE and the mixture model, and then fitting the mixture model to the FKE. If the AIC fails to reject, the $m+1$ model becomes the new mixture model and the procedure repeats.

Figure 4.26 illustrates the excess mass calculation. The mixture model (in this example, a single normal fit to the data) is shown with the corresponding filtered kernel estimator. Since the filtering mixture has a single component, the FKE reduces to a standard kernel estimator in this case. The difference curve is shown with the excess mass highlighted in gray. The larger (rightmost) region then determines the position, size, and shape of the new component added to the mixture. In this example, there were 100 observations drawn from the density $0.5N(0,1) + 0.5N(2, 0.25)$. Note that in this case, the region of largest excess mass is found near the component which is not modeled by the mixture. Thus, adding a term at the region of excess mass results in a better fit to the true density (in this case).

We must now choose the parameters associated with the new component. The idea is to use the center of mass, spread, and proportion of the mass represented by the excess mass to determine the new parameters.

Setting $\mathcal{R}$ to be the interval containing the largest excess mass and $e(x)$ the difference (excess mass) curve, we let

$$\begin{aligned} w &= \int_{\mathcal{R}} e(x)dx \\ \mu &= \frac{1}{w}\int_{\mathcal{R}} xe(x)dx \\ \nu &= \frac{1}{w}\int_{\mathcal{R}} (x-\mu)^2 e(x)dx \\ \hat{f}^{m+1}(x) &= (1-w)\hat{f}^m(x) + w\phi(x,\mu,\nu). \end{aligned}$$

The new mixture $\hat{f}^{m+1}$ is then fit to the kernel estimator to produce the new $m+1$ term mixture.

The mixture model selected by the AKMDE is shown in Figure 4.27. The top of the figure shows the density defined by the mixture model. The mixture components are depicted in the lower panel as a collection of dots and line segments, showing the means, standard deviations, and mixing proportions of the components. The x-axis corresponds to the mean of the component, the y-axis corresponds to the mixing coefficient, and the line segment denotes a one standard deviation spread about the mean.

In order to cluster the observations, an observation is assigned to the component with the greatest posterior probability. Alternatively, each observation could be given a "fuzzy" cluster designation consisting of the posterior probability vector. We consider only the first approach.

The result of the ADC algorithm is shown in Figure 4.28. The third component from the left, with a mean of about 0.08 and the smallest proportion of the first three components, has no observations in its cluster and so is not shown in the figure.

Looking at Figure 4.27, it is easy to discern three clear clusters, corresponding to the three obvious modes. Of course, some might argue for four, five, or more "obvious" modes. Looking at Figure 4.28, the first two images, corresponding to the first two components and the first mode in the density, are very similar. The next three or four images, corresponding roughly to the observations falling between 0.2 and 1.0, are quite similar yet show a progression from the third image on the top to the second image on the bottom. The last three images on the bottom correspond to the three components on the right, displaying at least two distinct clusters, with the middle one appearing to be a mixture of the two others. This kind of phenomenon is to be expected with this type of clustering.

Once the models were constructed, they were evaluated to determine whether they could be used for anomaly detection. There were 1,757,206 observations in the testing set. There were two basic experiments performed. In the first, each observation was given the probability associated with the machine/port pairing, according to the activity vector for that machine. If the probability exceeded a threshold, then the packet was considered "suspicious" and marked for further

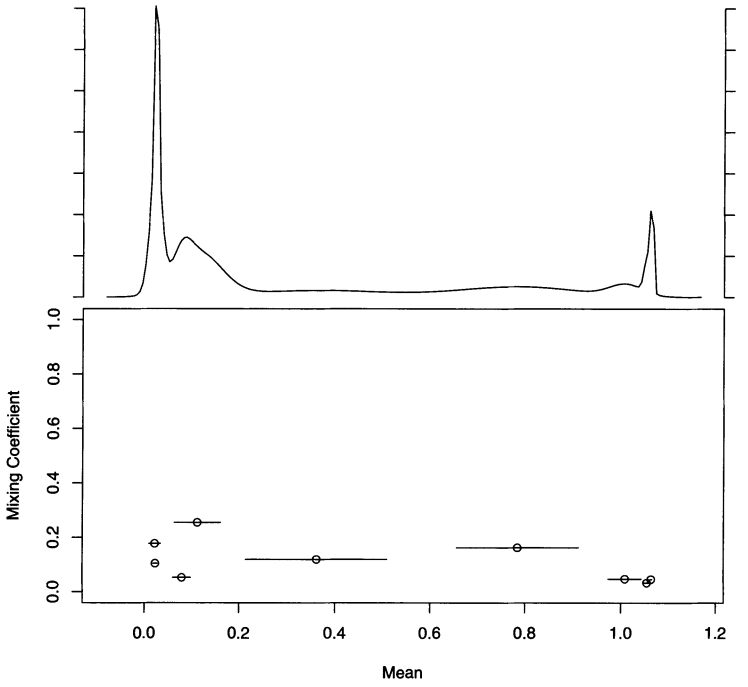

Fig. 4.27 Mixture model constructed using the AKMDE algorithm. The mixture model is plotted as a curve in the top plot, while the mixture components are depicted in the bottom, with the means of the components on the x-axis and the mixture proportions on the y-axis. The variance of each component is depicted via a two standard deviation bar centered at the component mean.

processing. Otherwise, the packet was deemed "normal" and ignored. For a variety of thresholds, the number of packets marked "suspicious" and the number of attacks detected as "suspicious" are tallied in Table 4.11.

The downside to using the individual activity vectors is the amount of storage required (or the time needed to access the disk for each packet). The storage increases linearly with the number of machines on the network. It also increases with the number of individual ports tallied. The speed and load of the monitored network will determine the time that can be devoted to making a decision about an individual packet.

The second experiment involved first clustering the machines as described earlier, using either the k-means or ADC method, and taking the profile for the machines in the cluster to be the cluster center. The same procedure was now used to determine whether a packet was "normal," using this profile rather than the individual activity vector. The results are shown in Tables 4.12 and 4.13. Since the clusters require the retention of only the cluster centers and a cluster assignment

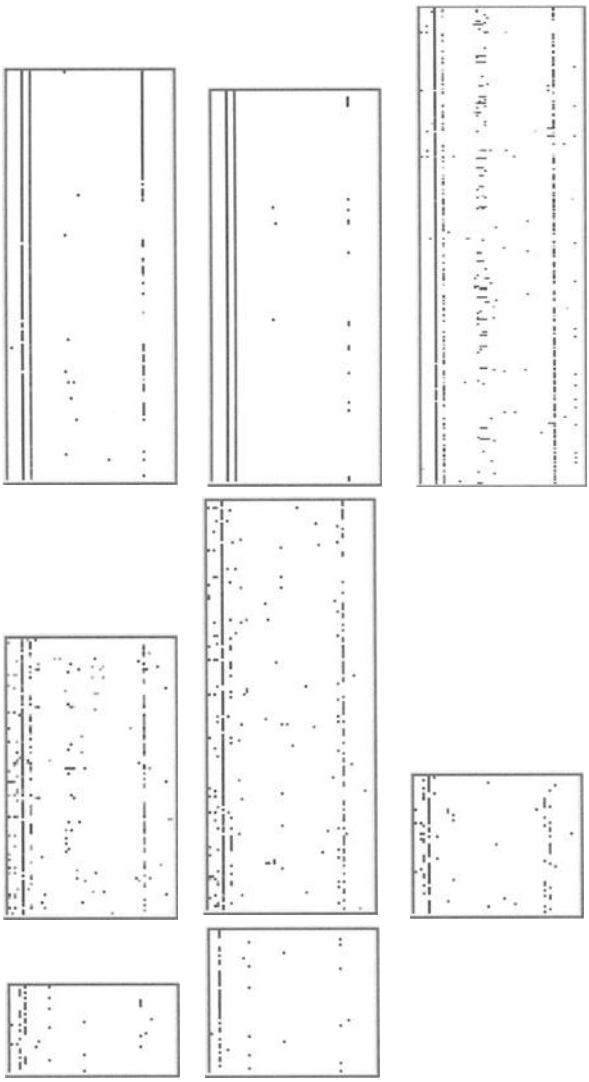

Fig. 4.28 Color histogram of the clusters from the ADC algorithm. Reading from left to right, top to bottom, these are in the order of increasing mean of the components.

vector, the processing/storage required is much less and is relatively insensitive to the number of machines monitored.

To get a better picture of the performance, a set of curves similar to ROC curves was produced. The percentage of the packets retained as "suspicious" is plotted against the number of attacks detected. This is roughly analogous to the probability of false alarm vs. probability of detection of the ROC curve. These are depicted in Figure 4.29.

As can be seen in Figure 4.29, the individual profiles are superior to the others for low thresholds, provided the criterion is a low number of attacks missed. This

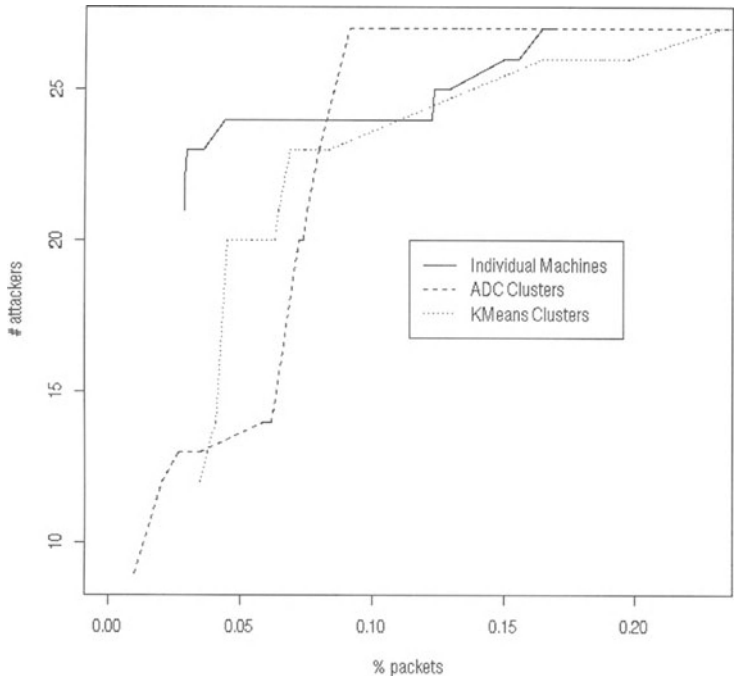

Fig. 4.29 Performance curves for the three techniques for activity level profiling. The precentage of packets determined to be "abnormal" is plotted against the number of attacks detected in the training data.

makes sense since the clustering of machines can cause activity that is rare for one machine to be considered common for the cluster, and hence this type of attack on the machine would be ignored at low thresholds. On the other hand, the ADC clusters did a better job if the criterion was to detect all the attacks. This is a result of the same kind of effect described earlier. Activities that are fairly rare for a given machine, can become much rarer when combined into an average profile through clustering, so attacks that previously required a larger threshold can now be detected at a lower one.

Table 4.11 Results of the profiling test using unclustered profiles.

Threshold	Number of Records	Number of Attacks Detected	Type of Attacks Missed
0	50,217	21	1 Telnet, 2 netbios, FTP, NFS, 1 misc
0.0001	50,288	22	1 Telnet, 2 netbios, NFS, 1 misc
0.001	54,069	23	2 netbios, NFS, 1 misc
0.005	58,962	23	2 netbios, NFS, 1 misc
0.01	63,410	23	2 netbios, NFS, 1 misc

Table 4.12 Results of the profiling test using ADC clustered profiles.

Threshold	Number of Records	Number of Attacks Detected	Type of Attacks Missed
0	17,069	9	Telnets, netbios, news, FTP, finger, tracerout, misc
0.0001	60,975	13	Telnets, netbios, news, FTP
0.001	108,529	14	Telnets, netbios, FTP
0.005	140,435	23	3 netbios, FTP
0.01	160,875	27	none

Table 4.13 Results of the profiling test using k-means clustered profiles.

Threshold	Number of Records	Number of Attacks Detected	Type of Attacks Missed
0	61,023	12	Telnets, netbios, FTP, misc
0.0001	78,642	20	nebios, FTP
0.001	112,961	21	nebios
0.005	131,393	23	4 netbios
0.01	146,742	23	4 netbios

The conclusion is that this technique can be used to filter out about 90% of the packets so that more sophisticated techniques can be used to focus on the 10% that may be suspicious. Since the processing/storage requirements are small for the clusters as compared with the individual machine activity vectors, the cluster techniques may be desirable for networks with a large number of machines.

Another consideration is the security policy of the network. Obviously, a network in which a very tight policy is enforced is likely to have machines clustering quite nicely along their activity profiles. A network with a loose policy, for example an ISP or university network, might have machines that do not fit any clustering scheme and, in fact, the individual activities might change according to which users are on the system. Thus, the approach described here is only a first step.

The "personal SHADOW" approach described on page 128 is similar to the activity profile approach described earlier. It places the processing on the individual machines, thus distributing the work across the entire network. This is a solution to the processing/storage dilemma described above. The technique appears to work quite well, in combination with an activity vector to aid in the definition of the filter used. Only those packets that do not match the filter's definition of "normal" are passed to the operator for consideration. This approach has been running quite successfully on my machine for several months now.

There are two major downsides to this approach, however. It requires a fair bit of maintenance to handle new situations. For example, when a new machine is added to our group, the machine needs to be added to the filter. Also, since only the suspicious packets are identified, the context is lost. Thus, for example, one cannot tell whether UDP packets are the result of activity initiated by the local machine.

Such packets, assuming the initiating activity is authorized, are probably not a problem. Packets coming unasked may be a probe, and hence should be treated with some suspicion.

4.5.5 Functional Data

We have considered network data both from the perspective of single packets (for example building filters to detect "bad packets" or detecting "anomalous" packets), and we have looked at aggregates such as the number of packets of a specific type in an hour. Now, we will briefly consider more general patterns of packets.

In a very real sense, network traffic really should be thought of in terms of collections of packets in time. This is most obviously seen in the TCP sessions, but it is also relevant for clustering machine activity types as was done in Section 4.5.

One way of approaching a more general theory of network modeling is through time series analysis. We will consider a slightly more general approach to the analysis of these data and look at functional data analysis (Ramsay and Silverman [1997]).

Consider the plots in Figure 4.30. This depicts the ten most active mail servers for a given network, over a period slightly longer than ten days (actually 250 hours). Time has been discretized into one-hour bins, and the number of SYN/ACK packets leaving the machine is tallied for each hour. This corresponds to the number of email sessions initiated within an hour. This does not correspond to the number of emails received since a single session can result in the transfer of multiple individual email messages. The first day of these plots is a Thursday.

Several things are readily apparent from the figure. First, there is quite a large variability of activity within each machine across time. Consider the first two and last four plots. These are machines that appear to have been active for only a portion of the ten-day period. Similarly, something interesting seems to have happened to the fourth and fifth machines during the last three days of the collection period.

Another interesting observation is the different dynamic ranges for the machines. The first six plots show values that go into the thousands or tens of thousands of sessions per hour while the last four get tens of connections per hour. Even though the pattern for, say, the first plot in the figure is very similar to the one for the last two plots, there is a significant difference between the activities of these machines.

Also, it is clear that there is periodic structure to some (but not all) of the machines. Consider in particular the third machine. There is a very strong periodic structure in this plot, especially if one ignores the outlier occurring on day 6 (a Tuesday).

The data for the third machine are replotted in Figure 4.31, where each day is plotted as a separate curve. Now, we can clearly see that the early morning and late night activity is quite stable over these ten days. Most of the variance of these curves occurs between 9 in the morning and 8 at night.

To further analyze these data, we replot, in Figure 4.32, the data with the sixth day missing. The mid-day variability is quite apparent in this plot, as is the periodic structure, with period approximately two hours. Figure 4.33 depicts the mean for

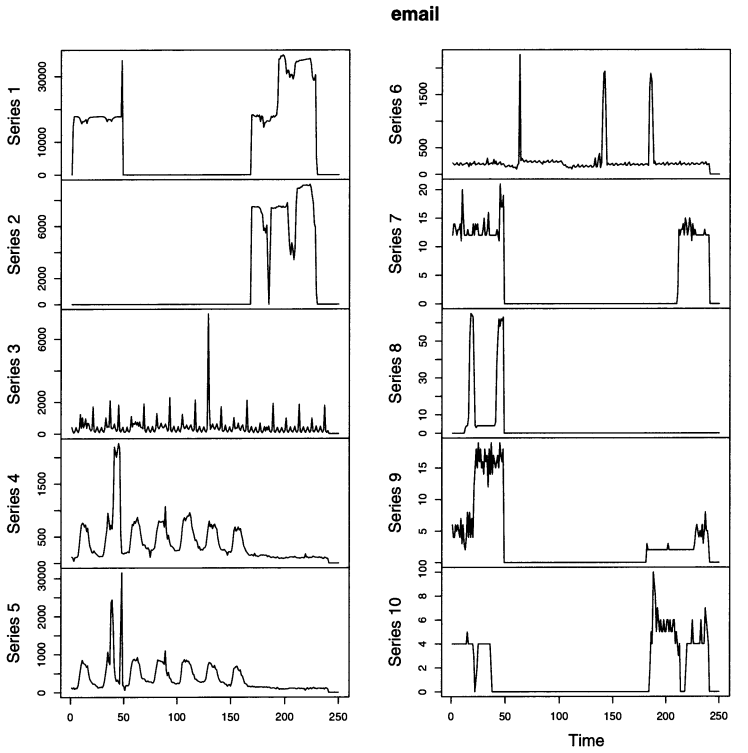

Fig. 4.30 Time series plots for the ten most active mail servers during a ten-day period.

these data, with dotted curves showing one standard deviation from the mean. The top graph depicts the full data set, while the bottom shows the graph with the sixth day removed. The mid-day variability is quite clear in these graphs, as is the fact that the nighttime variability is relatively low. Note that the bottom curve (with the outlying day 6 removed) shows clearly that the variance ramps up starting in the morning, peaks around 1 p.m. and then drops off into the evening. This information can be used to adjust thresholds for detecting an abnormal amount of activity on the server, which could be an indication of a spam or mailbomb attack.

4.6 EMERALD

EMERALD (Event Monitoring Enabling Responses to Live Disturbances) (Porras and Neumann [1997]), is SRI's environment for scalable, distributed intrusion detection and network monitoring. It is a hierarchical model, allowing different types of processing at different levels of abstraction. It is also highly modular, allowing different kinds of processing and analysis on different platforms or sections of the network.

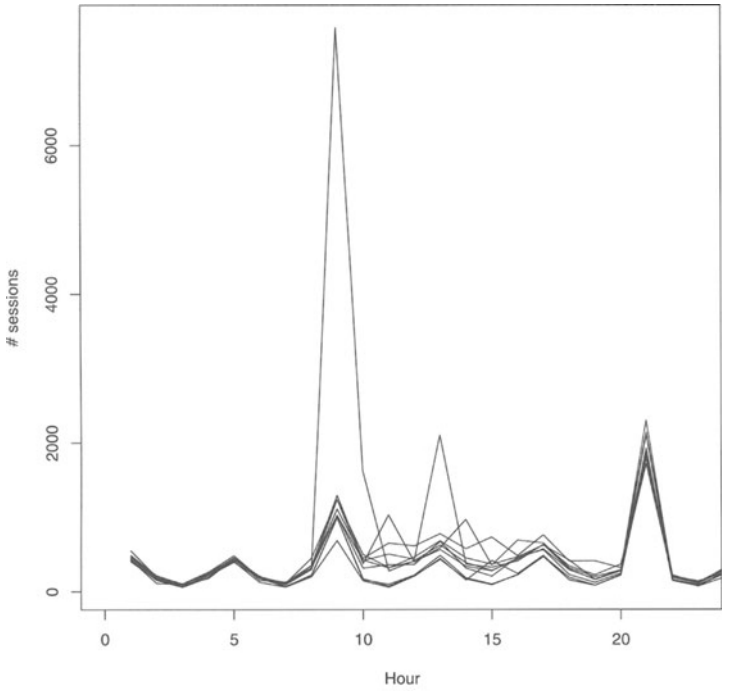

Fig. 4.31 Time series plots for the third most active mail server during a ten-day period. Each day is plotted as a separate curve.

Information on EMERALD can be obtained from many of the latest books on intrusion detection, such as Escamilla [1998], Amoroso [1999], and Bace [2000]. The definitive references are Porras and Neumann [1997] and the technical reports available at the SRI Web site (see Appendix D).

An underlying philosophy of EMERALD is to abstract the computation engines away from the details of the data or problem domain. This allows very flexible and extensible modules to be developed from a basic underlying architecture.

The EMERALD architecture is made up of a single basic unit, the monitor. A monitor can be thought of as a single IDS sitting on a specific host, but it is a much more general construct than this.

EMERALD is made up of three basic levels of processing. The service monitors are in the lowest level. These are the basic intrusion detection engines that monitor a host or small network. They communicate with other service monitors and with the next level, the domain-wide monitors.

The domain monitors correlate the reports from the service monitors. In keeping with the EMERALD philosophy, they are made of the same basic components as the service monitors. Only the specifics of the algorithms used in the analysis and

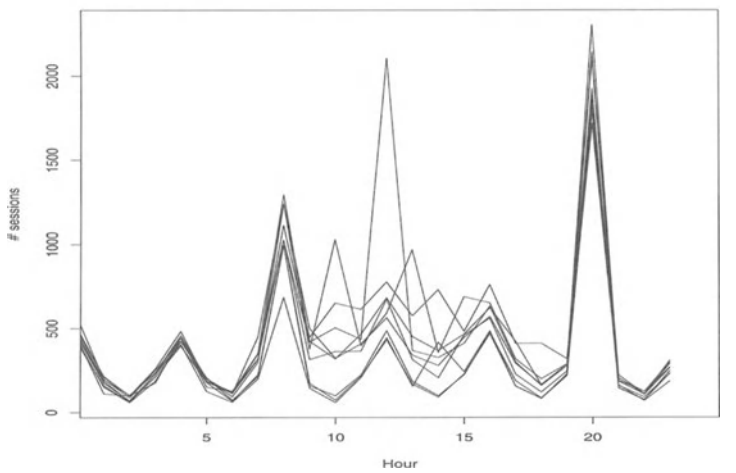

Fig. 4.32 Time series plots for the mail server in Figure 4.32 with the sixth day removed.

reporting changes. The domain monitors report to the highest level, the enterprise-wide monitors.

One way to think about the EMERALD hierarchy is in terms of a large corporation. The service monitors may be host-based security monitors on each desktop or network monitors on local area networks. The domain monitors correspond to

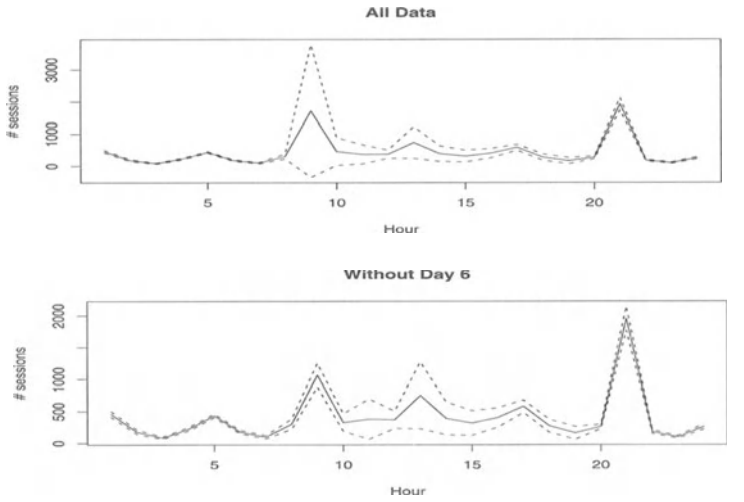

Fig. 4.33 Time series plots for the mail server in Figure 4.32. The plots show the mean and one standard deviation bands for the number of connections for the full data (top) and for the data with the sixth day removed (bottom).

the systems used by the security officers at each corporate site. These correlate the reports from the internal monitors and give the security officer a picture of the overall security at the site. The enterprise-wide monitors correspond to the systems at corporate headquarters, to which all the sites report, giving a picture of the situation for the entire company.

The generic EMERALD monitor consists of four parts:

- resource object,
- resolver,
- profiler,
- signature engine.

The signature engine is an analog of a set of SHADOW filters or a snort ruleset. This takes a set of rules for defining signatures and comparing data to signatures, making it easily configurable for different situations. This allows the detection of known attacks.

The profiler is the statistical anomaly detector. It uses NIDES, the "Next-generation Intrusion Detection Expert System" (Section 5.3). More properly, it uses the basic ideas of NIDES in a generic statistical profiling framework. This allows the incorporation of new techniques and different data types without requiring a redesign of the overall system.

The resolver is the coordinator and interface to other monitors and IDS systems. It correlates the results from the profiler and signature engine. It communicates any detections to the higher levels and/or the security officer.

The resource object contains the specific information needed for a particular deployment of the monitor. It contains all the information about the data feeds, rule sets, and so on, that the other parts need to perform their function. This is the single part of the monitor that needs to be configured for any deployment.

The resource object has a number of configurable components. These are implemented through pluggable libraries, allowing extreme flexibility in the functionality of an individual monitor.

The data streams used by the monitor are configured via the "event structures" that define the types of data that the monitor will process. This includes both the inputs and the outputs of the monitor. Related to the event structures are event collection methods, which define the basic routines for collecting and filtering the data streams.

The detection engines and analysis units are configured within the resource object. These define the intrusion detection algorithms implemented by the monitor.

Finally, there are communications configurations, called subscription lists, which define how communication between monitors will be handled. This is more than just a list of the other monitors, but it handles any information related to encryption and, in principle, multilevel security that might be implemented. There are also response methods defined, which determines what the monitor is to do once it detects an event.

While the EMERALD software is provided as an IDS, it is really a very flexible architecture which can be used to implement an IDS. Thus, it is not constrained by the actual implementation provided in the distribution.

4.7 WATCHERS

WATCHERS, which stands for "Watching for Anomalies in Transit Conversation: a Heuristic for Ensuring Router Security," is a distributed network monitor that watches for evidence of malicious routers, and removes them from the network. A malicious router is defined to be one that either discards packets or misroutes them (sends them on non-optimal routes). WATCHERS assumes that neighboring routers share the same view of the network, share a bi-directional link through which they can communicate, and that all (non-malicious) routers send packets along the shortest route (unless specifically directed otherwise).

WATCHERS works by having each pair of routers keep a set of counters that keep track of the packets that pass between them, either generated by one of them or forwarded from one to the other. Periodically, the routers report their counter values, and the counters are analyzed.

The main analysis consists of determining whether the packet flow is conserved: that is, that the number of packets going into a router is roughly the same as the number coming out.

This work is discussed in detail in Hughes [2000], Hughes et al. [2000], and Bradley et al. [1998].

4.8 GRIDS

The Graph-Based Intrusion Detection System (GrIDS) (Cheung et al. [1999]) is a program developed by the computer science department of the University of California at Davis. It is designed for use on large networks, analyzing network traffic in a hierarchical manner, that allows the technique to scale up to very large networks.

The idea is to construct and analyze activity graphs. The simplest version of an activity graph is a graph describing which hosts are connected to which. These graphs can be aggregated to allow various levels of resolution. Rules are constructed to detect "bad" or anomalous graphs, indicating potential attacks.

For example, consider the spread of a worm through a network (see Section 6.7 for more discussion about computer worms). If we were to represent the connections between machines as a graph, we might see something like Figure 4.34 in a very short period of time. This tree-like structure for the activity on the network is an indication of the spread of a worm.

To the GrIDS system, a graph is then a collection of nodes and edges, where both have attributes (for example, connection type (port), operating system, etc.). GrIDS contains a collection of rules for building and combining activity graphs and for analyzing them to detect intrusions, attacks, and anomalies.

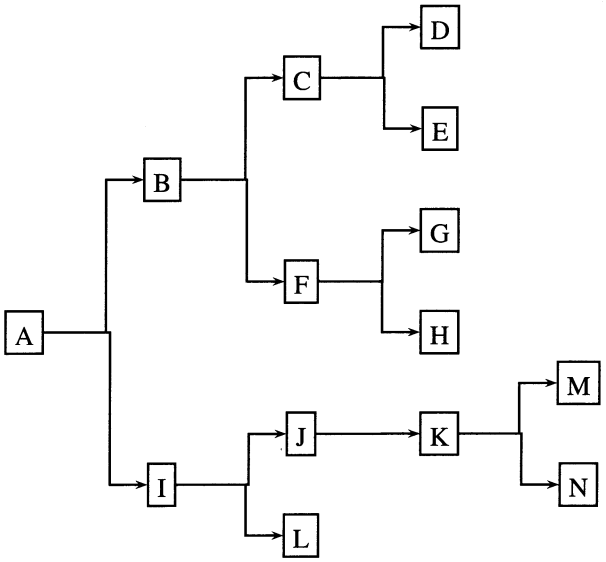

Fig. 4.34 A GrIDS activity tree for a worm. Each host (indicated by a lettered box) transmits the worm to other hosts, spreading the worm across the network.

4.9 MISCELLANEOUS UTILITIES

Here are some utilities that are useful for collecting data on a network. As always, care should be taken to ensure that the security policy allows the use of these utilities and that permission for their use has been granted. These are by no means a complete list. Resources for these and other utilities can be found in Appendix D.

4.9.1 nmap

The nmap program is a very powerful tool for security analysis. It performs a wide variety of scans on a system or network to detect open ports and potential vulnerabilities. It is also an extremely useful tool for an attacker.

Nmap operates by sending various packets to the host or hosts and seeing what comes back. In its simplest instantiation, it sends packets to a list of ports to determine what services are active. It can do this by actually trying to make a connection to the port (which can be easily detected since it will (usually) show up in the system logs), or it can use more stealthy techniques, such as sending only the SYN flag, the so-called "half-open" or SYN scan.

It can also send packets with strange flag combinations (for example, SYN and FIN both set). The purpose of these is to see how the host will react. Different operating systems will react differently to "illegal" packets, and this can allow one to determine the operating system of the host. This is called operating system fingerprinting.

The results of a simple scan against a machine follows. The name and IP address have been changed. Note that the version of nmap used in this example is not the most current (at the time of this writing), so newer versions may provide more or better information. In this case, the command used was

nmap -sS -v -O waldo

```
Starting nmap V. 2.02 by Fyodor (fyodor@dhp.com,
     www.insecure.org/nmap/)
Host waldo (10.10.12.193) appears to be up ... good.
Initiating SYN half-open stealth scan against waldo
     (10.10.12.193)
Adding TCP port 25 (state Open).
Adding TCP port 13 (state Open).
Adding TCP port 1024 (state Open).
Adding TCP port 9 (state Open).
Adding TCP port 111 (state Open).
Adding TCP port 513 (state Open).
Adding TCP port 515 (state Open).
Adding TCP port 80 (state Open).
Adding TCP port 21 (state Open).
Adding TCP port 22 (state Open).
Adding TCP port 37 (state Open).
Adding TCP port 514 (state Open).
Adding TCP port 841 (state Open).
Adding TCP port 1 (state Open).
Adding TCP port 23 (state Open).
The SYN scan took 0 seconds to scan 1068 ports.
For OSScan assuming that port 1 is open and port
     31200 is closed and neither are firewalled
Interesting ports on waldo (10.10.12.193):
Port    State       Protocol  Service
1       open        tcp        tcpmux
9       open        tcp        discard
13      open        tcp        daytime
21      open        tcp        ftp
22      open        tcp        ssh
23      open        tcp        telnet
25      open        tcp        smtp
37      open        tcp        time
80      open        tcp        www
111     open        tcp        sunrpc
513     open        tcp        login
514     open        tcp        shell
515     open        tcp        printer
841     open        tcp        unknown
1024    open        tcp        unknown
```

```
TCP Sequence Prediction: Class=trivial time dependency
Difficulty=26 (Easy)

Sequence numbers: 799148C0 799229C0 799339A0 79943580
                  79952B20 79961440
Remote operating system guess: IRIX 6.4 - 6.5

Nmap run completed - 1 IP address (1 host up) scanned
in 1 second
```

We see the open ports with service names (where known) next to them. This is useful for determining potential vulnerabilities. Note that it also tried to predict sequence numbers to see whether the system might be vulnerable to a TCP hijacking attack (see Section 4.3.3.2). In this case, it decided that it would be easy to guess sequence numbers (a potentially bad sign). Finally, it guessed (correctly) that the operating system was SGI's IRIX version 6.4 or 6.5 (it is actually 6.4 in this case).

The tcpdump trace of this attack follows. This is a subset of the actual trace, showing only the incoming packets. I have edited this to remove some redundancy (there were a total of 1093 packets incoming as a result of this scan). I have also annotated a few of the interesting lines, indicating these with a # sign at the beginning. The attacking machine is called "attacker." I have removed some information (such as sequence numbers) from the traces in order to conserve space.

```
12:55:46.078119 attacker > waldo: icmp: echo request
# Ping waldo to find out if the machine is up
12:55:46.156 attacker.52498 > waldo.161: S win 4096
12:55:46.156 attacker.52498 > waldo.122: S win 4096
12:55:46.156 attacker.52498 > waldo.2003: S win 4096
12:55:46.156 attacker.52498 > waldo.290: S win 4096
12:55:46.156 attacker.52498 > waldo.665: S win 4096
12:55:46.158 attacker.52498 > waldo.time: S win 4096
12:55:46.158 attacker.52498 > waldo.252: S win 4096
12:55:46.158 attacker.52498 > waldo.ftp: S win 4096
# Scan a few common ports. There were quite a few more
# packets like these.
12:55:46.384 attacker.52498 > waldo.412: S win 4096
12:55:46.384 attacker.52498 > waldo.1813: S win 4096
12:55:46.384 attacker.52498 > waldo.493: S win 4096
12:55:46.390 attacker.52498 > waldo.smtp: R win 0
# Note that a reset was sent to smtp (port 25, email)
# indicating that the port is open. This machine is
# running sendmail!
12:55:46.422 attacker.52505 > waldo.tcpmux: S win 4096
<wscale 10,nop,mss 265,timestamp 1061109567[|tcp]>
    Port 1 (tcpmux)
```

```
# This is the start of the fingerprinting.
# is used, as is port 31200. The first is an open
# port, the second one that is not open. These will
# react differently to packets sent to them. Note
# that nmap has also added a few options on for good
# measure.
12:55:46.422 attacker.52506 > waldo.tcpmux: . win 4096
<wscale 10,nop,mss 265,timestamp 1061109567[|tcp]>
12:55:46.422 attacker.52507 > waldo.tcpmux: SFP win
    4096 urg 0 <wscale 10,nop,mss 265,timestamp
1061109567[|tcp]>
# Note that the SFPU flags are set. This would never
# happen in normal traffic.
12:55:46.422 attacker.52508 > waldo.tcpmux: . ack 0
    win 4096 <wscale 10,nop,mss 265,timestamp
1061109567[|tcp]>
12:55:46.422 attacker.52509 > waldo.31200: S win 4096
<wscale 10,nop,mss 265,timestamp 1061109567[|tcp]>
12:55:46.422 attacker.52510 > waldo.31200: . ack 0 win
    4096 <wscale 10,nop,mss 265,timestamp 1061109567
[|tcp]>
12:55:46.423 attacker.52511 > waldo.31200: FP win 4096
    urg 0 <wscale 10,nop,mss 265,timestamp 1061109567
[|tcp]>
12:55:46.424 attacker.52498 > waldo.31200: udp 300
# Now try to guess sequence numbers.
12:55:46.424 attacker.52505 > waldo.tcpmux: R win 0
12:55:46.424 attacker.52507 > waldo.tcpmux: R win 0
12:55:46.705 attacker.52499 > waldo.tcpmux: S win 4096
12:55:46.706 attacker.52499 > waldo.tcpmux: R win 0
12:55:46.735 attacker.52500 > waldo.tcpmux: S win 4096
12:55:46.736 attacker.52500 > waldo.tcpmux: R win 0
12:55:46.765 attacker.52501 > waldo.tcpmux: S win 4096
12:55:46.766 attacker.52501 > waldo.tcpmux: R win 0
12:55:46.795 attacker.52502 > waldo.tcpmux: S win 4096
12:55:46.796 attacker.52502 > waldo.tcpmux: R win 0
12:55:46.825 attacker.52503 > waldo.tcpmux: S win 4096
12:55:46.826 attacker.52503 > waldo.tcpmux: R win 0
12:55:46.855 attacker.52504 > waldo.tcpmux: S win 4096
12:55:46.856 attacker.52504 > waldo.tcpmux: R win 0
```

There are a large number of options available for nmap. Some of the useful ones are:

- **-sT** TCP connect port scan. One need not be root to execute this option. This completes the three-way handshake to those ports that are open.

- **-sS** TCP stealth SYN port scan. This sends packets with only the SYN flag set. Those ports that respond are then sent a reset packet to close off the connection.
- **-sF** TCP stealth FIN port scan. Also, using X or N in place of F will result in an Xmas scan (all flags set) or Null scan (no flags set).
- **-sU** UDP port scan.
- **-O** Use fingerprinting to determine the operating system.
- **-F** Fast scan (only scan those services listed in /etc/services).
- **-o** logfile Output results to a logfile.
- **-g** port Set the source port number for the scans.
- **-v** Be verbose in output. Can be given twice for even more information.
- **-h** Print help.
- **-V** Print version information.

There are a number of other options available. See the man page for more information. Some of these are better suited for using nmap as an attack tool rather than a vulnerability scanner to improve security. One such is the -D option, which allows one to add decoy hosts into the scan. The result is a scan that appears to come from a number of hosts, making it difficult to determine who the attacker really is.

The decoy option provides an opportunity for the statistician. Given packets apparently from several sites (as generated by nmap), can one determine which is the real attacker? This depends on how the packets were generated. Let us consider a couple of cases as an illustration.

If the packets from the different sites are identical, one would have to use other information to try to determine which site is the real one. For example, one could do an nslookup and a whois to determine the names and (rough) locations of the machines. This could allow one to estimate the approximate number of hops taken. This, with an estimate of the initial TTL value (which can be obtained from the operating system (OS) estimate, by p0f (Section 4.9.2), or by using techniques discussed in Section 4.3.2.2), can be used to see which of the packets is most likely to have originated from the attacking machine and which are decoys.

If all packets are different, one could use passive fingerprinting to determine the operating systems for the packets. Using statistical models for the operating systems would allow one to perform a goodness of fit with the different operating systems to see which of the packet streams best fits one coming from the purported OS.

Although it may seem that nmap is purely an information gathering tool, it is not without its dangers. A colleague of mine was doing a scan of our network when one of the routers he was scanning went down. It turned out there was a bug in the router software that caused it to be vulnerable to a particular type of

packet. The resulting effort to find and fix the problem resulted in the network being essentially shut down for more than a day. Although one can argue that it was not my colleague's fault (and he did indeed argue this), it is clear that one should never institute scans against computers without first obtaining permission from all involved, including the network security officer (fortunately, my colleague did have permission for his scan).

Nmap may be obtained from www.insecure.org/nmap.

4.9.2 p0f

Although nmap can do a good job of determining the operating system of a remote host, it is an active system, which means that the host can be aware of the fingerprinting attempt, and firewalls can block the attempt. Imagine instead that you are attacked by a system, and you wish to determine the operating system of the attacking system (I will leave the issue of why you might want this information to your imagination). If you run nmap against the attacker, you alert him/her that you have detected the attack. It would be nice if you could tell the operating system simply from the incoming packets. This is what p0f attempts to do.

P0f operates by considering incoming SYN packets and extracting information from the packet to be used to characterize the operation system. For example, the time-to-live (TTL) value, window size, maximum segment size, whether the don't fragment flag is set, and which options are used can tell a great deal about the operating system of the source machine.

An example of p0f output is

10.10.10.23 [15 hops]: Linux 2.2.14 or Cobalt Linux 2.2.12C3

This was run on a packet from a machine that attacked my work machine (as always, the IP address has been changed). Whether or not the machine really is running Linux is unknown (I resisted the temptation to run nmap on it and find out).

Note that p0f uses the TTL to determine the operating system. However, this is inherently unknowable since it has been decremented by an unknown number of routers in transit. P0f tries to guess the value by looking for reasonable initial values for known operating systems and matching the other parameters up with the operating system. Thus, one obtains both an estimate of operating system type and an estimate of distance away (in terms of the number of hops taken).

The version of p0f that I have looks only at the SYN packet. Adding the other protocols, and information about the TCP session, would be a useful enterprise. Also, it uses a table lookup. Adding statistical fingerprinting would be a very interesting endeavor.

The p0f program can be obtained from http://lcamtuf.hack.pl.

4.10 FURTHER READING

A number of papers have been written on the topic of network intrusion detection at the level appropriate for the layperson. See, for example, Herringshaw [1997], Mukherjee et al. [1994], and Meinel [1998].

Another statistical modeling technique is discussed in Cabrera et al. [2000] in which a Kolmogorov-Smirnov test is used to detect deviations from "normal" traffic activity. A neural network approach is discussed in Tan and Collie [1997].

Girardin [1999] proposes using Kohonen maps (Kohonen [1995] or Van Hulle [2000]) to visualize network activity. Since these maps also can be used for clustering, this is a potential alternative to the work in Section 4.5.

Some comments on experience developing and using EMERALD are given in Neumann and Porras [1999].

A discussion of a methodology for avoiding network-based denial-of-service attacks is found in Meadows and McLean [1999]. It is argued that the defense against DOS attacks must be built into the protocols themselves.

An agent-based technique for attacking networks is discussed in Stewart [1999]. This argues that much more sophisticated detection techniques are needed to detect the attacks of the future.

A technique for using finite-state machines for the detection of intrusions is discussed in Vigna and Kemmerer [1998]. Several spoofing attacks are described, and details on how they could be detected and analyzed using these methods are discussed. Other finite-state machines are discussed in Chapter 4 of Bace [2000].

Sekar et al. [1999b] describe a language for specifying normal and abnormal packet sequences. This results in a concise and efficient mechanism for specifying both normal activity and specific types of attacks that use abnormal packets. A sufficiently well-designed set of specifications should be able to detect any attacks that use malformed packets, such as Targa3 or teardrop (Sections 4.3.1.5 and 4.3.1.8), and can also detect floods and scans and other packet activity that is not normal for the network.

A different kind of traffic analysis is discussed in Ettema and Timmermans [1997]. This looks at analysis of travel patterns. Some of this may be relevant to routing or anomaly detection in networks. Newman-Wolfe and Venkatraman [1991] discusses techniques to prevent traffic analysis. The less an attacker can learn about the typical patterns of traffic on your network, the less they learn about the machines and users on the system. For example, knowing the traffic patterns can indicate the kinds of applications running on the different machines, as described in Section 4.5.3. The basic idea is to use dummy packets, reroute, and delay packets in order to confuse any monitor as to the real patterns of activity on the network.

I have not discussed anonymity on the Internet except to note that packets can be spoofed. Chapter 5 of Amoroso [1999] contains a fairly extensive discussion of anonymity, including how to track back attackers to determine their identity. He also discusses many of the utilities for gathering information about people on the Internet. Chapter 7 discusses a number of techniques for trapping intruders. These should be used with caution but can be very useful tools for the security analyst.

Another area of Internet security that is missing from this book is cryptography. This is a huge field and far beyond our scope. Some people seem to think that encryption solves all security problems - that with a sufficiently sophisticated encryption scheme their networks would be safe. This is not the case, although it is certainly true that properly used encryption is a powerful tool for security. Rather than list a few of the hundreds of books on cryptography, I leave it to the reader to browse a local bookstore. Instead, I will mention an interesting new book (at the time of this writing) by Ryan and Schneider [2001]. This book suggests that security can be enhanced by properly modeling security protocols and using the models to suggest improvements or point out flaws. This book focuses primarily on cryptographic protocols, but the basic idea is sound throughout the security field.

One of the big areas of research, particularly among military organizations, is that of data fusion. The goal is to determine optimal ways of "fusing" data from disparate sensors into a common framework to improve detection and identification, situational assessment and analysis, and provide a unified picture of the battlefield. In Bass [2000], the techniques of data fusion are proposed as a set of tools that should be applied to the intrusion detection arena.

Part III

Viruses and Other Creatures

5

Host Monitoring

5.1 INTRODUCTION

Host monitoring refers to gathering and analyzing information related to the security of a single computer. This usually involves looking at the security log files, monitoring processes, disk usage, file access, and other information related to the proper functioning of the computer. It can also refer to monitoring users on a computer, in an attempt to detect unauthorized users.

A good reference for host-based attacks is Kendall [1999]. We will cover the main attacks described in this thesis as well as several from other sources.

As with network monitoring there are denial-of-service attacks that are focused on attacking a specific host and that make use of application flaws or quirks rather than network intricacies. These are of essentially three main types. They either attempt to bring down the machine, bring down an application, or destroy data. We will see examples of all three in this chapter.

In addition, we will see two new classes of attacks: the so-called "remote to user," in which an attacker gains access to the machine from outside; and "user to root," in which the attacker gains super user permissions. In this latter case, the attacker can eliminate all (local) evidence of the attack, obtain any information (not protected by encryption or other methods), or remove any files on the machine.

5.2 COMMON ATTACKS

5.2.1 DOS

We have seen a number of denial-of-service attacks from a network perspective in Section 4.3.1. Now, we consider some that are more properly grouped with host-based attacks. These are generally attacks that deny access to a service or a machine by exploiting vulnerabilities of particular applications rather than by attacking the IP stack or blocking access to the network.

5.2.1.1 Apache2 Old versions of the Apache Web server can be slowed to a crawl or caused to crash by sending many requests with a large number of HTTP headers. Typical HTTP requests contain less than 20 headers, whereas an Apache2 attack will have requests containing thousands. This causes the load average of the machine to rise dramatically, memory usage to climb, and usually the machine will crash.

Obviously, in order to detect this kind of attack, the requests coming in to a Web server should be tracked. Some statistics worth knowing would be the number of connections per time unit (where the time depends on the typical load on the server), the number of headers per request, the number of distinct machines per time unit, and the number of requests per machine. Given an estimate of the typical variation, these statistics can be used to flag any large deviations from normal activity as being worthy of the security analyst's scrutiny.

More details about this attack can be found at

http://www.geek-girl.com/bugtraq/1998_3/0442.html

5.2.1.2 Back Another attack against old versions of the Apache Web server was the back attack. In this attack, requests were sent that contained a large number of front slashes '/', on the order of six or seven thousand. This causes a temporary slowdown of the machine. The machine recovers when the attack stops.

This is an example of what I call a "stupid user" attack. When an application is not designed to handle strange but technically legal input, it may be vulnerable to attack, for example, by someone simply holding down a key or even hitting the keyboard with one's forehead. Generally, it is a good idea to harden one's applications against "stupid users."

5.2.1.3 Mailbomb A mailbomb is an attack against an individual, which can also cause the machine to crash. The idea is to send many mail messages to a user on the machine. If "many" is in the hundreds, this can cause great pain to the individual. If "many" is thousands, and the mail messages are large, the mail queue can fill up and the machine crash. Also, the disk can fill up with these messages, causing other legitimate messages to be undeliverable or to be lost.

Mailbombs are easy to implement and quite popular. They are cousin to the other scourge of email, spam. Spam is the name for junk email mailings. They are to email what all those credit card solicitations are to the postal system (the postal system is also known as "snail mail"). A company (or individual) will send unsolicited email to a large number of recipients. Often, the sender field is spoofed in order to make it difficult to take action against the sender. If you

have ever received email directing you to porn sites on the Web, you have been spammed.

Note that although the sender (or "from") field can be spoofed (and often is), the originating IP address cannot be since email uses the TCP protocol. Thus, it is often possible to contact a system administrator responsible for the machine, who can detect and stop the offending party. Some of the larger ISPs do not respond to these requests, others are very responsive, so at the moment it is a hit-or-miss proposition to obtain relief from spam (or mailbombs).

Although spam is universally considered to be an evil, it is generally not an attack, and aside from a small amount of resources required to process the email, it does not constitute a threat. Mailbombs are a threat, since they can deny access to legitimate email, or to the machine itself.

5.2.1.4 Webbomb This is my terminology for the Web equivalent of a mailbomb. It is easy to write a script to generate a large number of Web requests to a single Web server. This is the poor man's version of the distributed attack (the distributed denial-of-service attack discussed in Section 7.5.1) that brought down many famous Web sites.

This can be as devastating to the Web server as a mailbomb can be to the mail server. Like the mailbomb, it is easy to detect (a lot of requests coming from the same site), and so can be defended against. Note that both email and Web use TCP, so one cannot spoof the IP address since the full handshake must be completed before the data are transfered. This is one reason why attackers will often first obtain access to an intermediary computer from which to mount their attack (called "looping") to make it difficult to track the attack back to the attacker.

5.2.1.5 Resource Hogging Any kind of program that hogs the resources on a machine can be a denial-of-service attack. These usually require user access to the machine. We will see a simple worm in Section 6.7.1, which will eventually bring a machine down. A similar idea is to run something like

```
#!/bin/csh
cd /tmp
while(true)
mkdir foo
cd foo
cp -r ~/* .
end
```

This is essentially the program given on page 249 of Escamilla [1998] (please do not try this or any of the "attacks" described in this book). The first line indicates that the program is a cshell program. This program will create a series of directories in the /tmp directory, filling each one with a copy of the user's home directory. This will quickly fill up the disk as well as cause some annoying slow downs due to all the copying. Even without the copy command (the line starting with "cp"), this will bring down most systems. Some systems are invulnerable to this attack because of disk quotas which make it impossible for a single user to use too much disk space.

The inverse of the resource hog is the famous

```
rm -r *
```

If executed in a directory in which the user has write privilege, all the files (and all files in subdirectories) are removed. This is the ultimate in denial of service. The only protection is regular backups.

Of course, nobody would do such a thing by accident, right? I'll bet that the rm command for root is aliased to "rm -i" on your system. This asks the user to confirm any file removals. It is extremely annoying, but, since removal is forever, it is a very good idea. (Helpful hint: using "\rm" executes the unaliased version of the command. Use with caution. The alias is there for a reason.) Most of us have the experience of typing quickly and hitting the return just before we see that instead of typing

```
rm *.bak
```

we've typed:

```
rm * bak
```

or some such. Instead of removing the backups that we thought we no longer needed, we've removed everything (including the backups).

It is a good idea to back your work up regularly. I tend to make "Save" directories where I put copies of things while I am working. The preceding command is not recursive, and so will leave the Save directory intact. This will not protect you from attackers, but will provide some protection from yourself.

5.2.1.6 Creative Telnets Old versions of Windows NT were vulnerable to telnets to high ports (anonymous [1997]). For example, if one telnets to port 1031 and sends a few characters, the destination machine will crash. This is a very old vulnerability and is undoubtedly fixed in newer versions of NT.

This points out one of the difficulties in computer security. Bugs appear in operating systems (and applications) all the time, and until someone discovers the bug, and thus describes the attack it allows, it is difficult to defend against the resulting attack. Open-source programs help here since people can look at the code and determine both the problem and the fix without waiting for the vendor. Of course, this is a two-edged sword. The potential attackers can also look at the code and determine new attacks.

5.2.2 Remote to User

In order to get into a computer, one must either have physical, network, or modem access. If the computer is on the Internet, network access is often easy to achieve unless it is protected by a particularly tight firewall. Physical access can be obtained by breaking-and-entering or simply by wandering around during business hours acting like someone who belongs there. Modem access can be obtained by "war

dialing." This is a process whereby a range of phone numbers are called, looking for numbers that are answered by machines. Surprisingly, or maybe not, many machines are configured to have the modem answer if its number is called. Most of the time the user is not aware of this "feature."

An attacker that wants to utilize your computer or gain access to information on the computer must somehow obtain a user's account. The simplest way is to simply log in with the correct user name and password. In order to do this, the attacker must obtain this information from somewhere or guess it.

Probably the most successful method for obtaining user and password information is through social engineering. Through various tricks, the attacker gets a legitimate user to provide the desired information. This is discussed at length in Hafner and Markoff [1995] and in Denning [1999], pp. 216–217.

A similar idea is to go through the trash (hence the name "trashing") of the victim organization, hoping to come across interesting and useful information. This too is discussed in Hafner and Markoff [1995] in some detail. Trashing is also referred to as "dumpster diving" (see also Denning [1999], pp 159–160).

An alternative approach is simply to try to guess the password. First the attacker obtains (or guesses) a user name. Some common ones are "guest," "lp," "root," and "administrator". If the attacker knows the names of some of the people in the victim organization (say, through "trashing"), user names can be guessed by performing simple operations on the names. For example, Diane B. Jones probably has a user name such as: djones, jonesdb, dbjones, or dianej.

Given a user name, the password can often be guessed because people often do not use secure passwords. The password for "guest" is often "guest." If Diane has a daughter, try her name, birthday, etc. It would seem that even with this kind of information, the number of possibilities is endless, but surprisingly this approach has been used quite effectively.

This kind of guessing can be detected by considering the number of access attempts (such as telnet or FTP) that fail. If the attacker is patient, these may be spread over a long time, making detection difficult. Also, it is possible to guess the password on the first try, so this is not a reliable approach to detection.

Another key to detection is to consider the source of the connection attempt. Attempts from unusual places, or at unusual times, are a tipoff that something suspicious may be happening. This requires some kind of user or activity profiling, such as is discussed in Sections 4.5 and 5.5. Many attacks at NSWC are detected because they come from foreign sources. Remember, however, that the apparent source is not necessarily the true source. Attackers often go through a number of intermediary machines before they attack a well-defended site in order to hide their trail and avoid prosecution. This is called "looping." In some countries, most "cyber-attacks" are legal, and therefore an attacker may first obtain a machine in one of these countries before attacking a site in a country with more restrictive laws.

If one can obtain the password file (/etc/passwd on a Unix machine), one can attempt to crack the passwords using one of the many password cracking programs. These run through a dictionary, trying all the words, and various modifications, as passwords, attempting to find one that works. Recall that the password file contains encrypted passwords, using a one-way encryption scheme, so the cracking program

encrypts the prospective password and compares it with the one in the password file.

There are many password cracking utilities. I like "John the Ripper," available at

http://www.openwall.com/john/.

There are a number of other such utilities, including some that can be built in to the password program, so that when users change their password an attempt is made to crack the new password immediately, thus (hopefully) catching easily cracked passwords before they are used.

Password cracking may take a while, especially if the dictionary is large and the cracking tool tries many variations (for example, replacing the letter "o" with the digit "0" or concatenating two small words together), but the attacker can run this at home, with no risk of detection, once the password file has been obtained. Attackers will also use other computers they have compromised to distribute the password cracking code. Thus, it is not a bad idea to watch for processes named things such as "crack" or "L0phtcrack" as indications that your machine has been compromised and is being used to compromise other machines.

Trying to crack a password file would at first seem to be as futile as trying to guess one (although even this latter is not always as futile as one might think). An informal investigation of password cracking is reported in Farmer and Venema [1993]. They checked 656 hosts and found that they were able to obtain 24 password files with little difficulty. Of these, a third had an account with no password. They ran crack, a freely available password cracker, and found that in a 10-minute run on 1594 accounts they were able to obtain more than 50 passwords. After a few days, they had:

- 5 root passwords.
- 19 files that had at least one password, giving them access to 80% of the machines.
- 259 passwords guessed.

If you think we are more secure now since everyone is now so security-conscious, you haven't been paying attention to the news lately (no matter when you read this!). For another anecdote: the first time I ran crack on my machine at work, I cracked a password (fortunately, not mine).

The password file can often be obtained without logging on to the machine, for example through the phf attack. In this, a machine with a Web server and the phf program in its "cgi-bin" directory is sent a request for phf to provide the password file, which it does. Obviously, if you have phf on your machine, you should remove it. Other cgi programs have this vulnerability, so it is a good idea to remove all such programs that you do not need and to check the security sites to make sure no new vulnerabilities have been discovered.

Some machines run "TFTP," the so-called "trivial file transfer protocol." Some of these have misconfigured the software to allow reading of files without a password, which is another way to obtain the password file. Some FTP servers have real password files in their /ftp/etc directory. Denning [1999] reports that at least

one individual was able to obtain encrypted root passwords simply by doing a Web search.

The preceding discussion points out the need to carefully secure your password file. Some systems use shadowed password files, where the world-readable password file contains no password information. The passwords are kept in a "shadow" password file with restricted access.

5.2.2.1 FTP Write This is a specific case of a general idea. If a user's home directory contains a ".rhosts" file, this file is checked for "trusted" remote hosts. This is a nice utility to make it possible to move from machine to machine without needing to type user name and password information. This is very useful, and loved by users, which is a tip-off that it is a huge security problem. If the ".rhosts" file contains the string "++," then any user from any machine can log into the account (using "rlogin") without providing a password.

Anonymous FTP is one of the most useful methods for providing file transfers across the Internet. Even most Web servers utilize this facility for transferring files that are not meant to be displayed as Web pages. A directory is set up (with protections to make it "locked off" from the rest of the directory structure) with a user name "anonymous" which can be used to FTP to the computer without a password (most anonymous servers require the password to be given as the email address of the person logging in; generally little to no checking is done on this). Only those files under this directory are accessible, so the system administrator places those files that are to be shared with the world in directories under this FTP directory.

The anonymous FTP directory should not be writable. If it is, the attacker can place a ".rhosts" file in the directory. The attacker then logs out and logs back in (using "rlogin" this time) as "ftp." The user "ftp" is not restricted in the same way as "anonymous" since the application used is now rlogin.

This attack can be detected either by searching the content of packets for the string "rhosts" or by using a file integrity checker such as tripwire (Section 5.6.6) to check that the FTP directory has not been modified.

This specific attack relies on a misconfigured FTP server. However, this basic idea works for any attack that allows the attacker to write a file in a user's home directory. For example, misconfigured TFTP servers have been known to allow anyone to write files without requiring a password.

5.2.2.2 Buffer Overflows There are a number of buffer overflow attacks that allow an outsider to gain access to a computer. Kendall [1999] discusses a number of them that utilize bugs in imap, named, and sendmail. The interested reader is encouraged to read Kendall [1999] for more details on these attacks.

These attacks are network-based in the sense that they operate by sending packets with particular data to an application running on a remote machine. The data contain executable commands that the application runs, giving the attacker access to the machine. We will discuss this in more detail in Section 5.2.3.1. The attacks can be detected by looking for specific strings in the content of packets; however, this will not allow one to detect new attacks against other applications.

Thus, the attacks really need to be detected by monitoring the host system, watching for inappropriate accesses, activity, and changes in access permission.

5.2.2.3 Trojans We will discuss trojan programs in more detail in Chapter 7, but there are a few trojan programs that are worth mentioning in this section. In a nutshell, trojan programs are programs that appear to be one thing but act to provide information to an attacker or mount the attack themselves.

In an X Windows environment, only those machines that the user has indicated have permission may connect to the X Windows server. Thus, if you are running X Windows as yourself, and su to root, you may find that you cannot execute commands that use the X console because you (the owner of the session) have not given root permission to do so. One solution to this is to type "xhost +machine," where "machine" is the name of the host you are on (this needs to be executed as the owner of the X console). Lazy (or inexperienced) users may choose to simply type "xhost ++". Like the "++" in the .rhosts file mentioned previously, this allows access to the console from any machine.

Assume that the attacker can gain access to the X console. The attacker can then run a fake "xlock" program that makes it look like the screen has been locked and request the password from the user to unlock the screen. Once the unsuspecting user has typed the password, the screen lock goes away and the user is none the wiser. However, the attacker now has the user's password.

Another way to gain information on an open X server is simply to watch all the characters typed. This can be done, if the X console is open, as above, providing the attacker with a lot of interesting information. For example, if the user logs in to another machine, the user name and password may be obtained.

5.2.3 User to Root

User to root attacks are ones designed to extend the user's privilege to that of the super user. These attacks can be mounted by attempting to obtain the root password, either by cracking the passwords in the password file, by sniffing the password, or by social engineering. These have been discussed earlier, so we will consider other attacks, which do not rely on obtaining the root password.

The most common user to root attack is the buffer overflow attack. In this, the attacker exploits a programming error that allows data to be placed on the execution stack. When the data are executed, the program corresponding to the data provides the user with root access.

Other attacks involve clever tricks that are not easy to characterize. One such attack, taken from Kendall [1999] works as follows (the attack works against SunOS 4.1).

The internal field separator (IFS) is used to define the character that separates fields. Generally, this is set to the space character. The attack is:

1. Copy /bin/sh to ./bin (The "./" corresponds to the current directory).

2. Add "." to the beginning of your path variable if it is not already there (this makes your current directory the first one searched for executable programs).

3. Change the IFS to "/".

4. Execute the command "loadmodule a" (this is a program that loads modules in the Sun operating system).

The loadmodule program executes the command

exec("/bin/a")

which gets translated (thanks to the IFS) to the equivalent of

exec("bin a")

which then executes the local copy of "bin," which happens to be a shell. Since loadmodule does its "exec" as root, the shell is executed as root.

This kind of trickery is quite common. It illustrates the level of knowledge required to come up with some of these attacks. While so-called "script kiddies" are held in contempt by security experts due to their lack of knowledge and the lack of sophistication of their attacks, the people who discover the attacks are generally quite knowledgeable. In fact, they are often the very security experts tasked with protecting the machines and networks. There is some controversy as to whether these people should publish the attacks they discover, but it is generally agreed that this is a good thing, allowing system administrators (and vendors) to fix the problems before someone with evil intent discovers and utilizes the attack.

5.2.3.1 Buffer Overflow Examples There are a number of buffer overflow attacks detailed in Kendall [1999]. Instead of detailing each one, let us consider a hypothetical attack in some detail. The interested reader is encouraged to read Kendall [1999] for some specific examples.

The basic requirement of all buffer overflow attacks is a program that places data into a buffer without doing any bounds checking to make sure that the data do not extend beyond the buffer. Data that overflow the buffer can overwrite the execution stack or parts of the program and hence be executed as if they were legitimate parts of the program. By careful manipulation of the data placed on the stack, the user can execute pretty much anything desired, with the same permission level as that of the targeted program. Thus, if the program is owned by root, the attacker's program is executed as root.

Figure 5.1 illustrates the attack. In this example, the attacker calls the program with a long string for an argument, which gets placed into a buffer. The buffer is too small to hold all the arguments, and the programmer did not bother to do bounds checking (after all, who is going to call this program with 100 arguments or more). The attacker has carefully crafted the argument string so that it corresponds to spawning a shell command with root privileges.

Ko et al. [1994] describe a buffer overflow attack on the finger daemon. The problem is that finger uses the library routine "gets" to read strings into a buffer, but the gets function does no bounds checking. Thus, the attacker, who knows the size of the buffer from investigating the finger code or by trial and error, can send

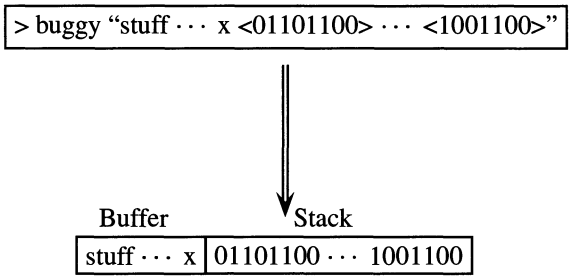

Fig. 5.1 An illustration of a buffer overflow attack. The attacker executes the command "buggy" with a very long string. The buggy code does not check the string size to ensure that it will fit in its buffer ("x" marks the end of the buffer in the figure), and as a result the string overflows into other memory, in this case the execution stack. The attacker has arranged that the binary values of the string that gets written on stack memory are the instruction code for some action, such as spawning a terminal.

a sufficiently long string with appropriate binary code to be placed on the stack, and this code is then executed.

There are a number of ways to detect such an attack, depending on the sophistication of the attacker. If source code for an implementation has been captured (or downloaded from a Web site), a signature can be constructed to look for unique strings that appear in the code. Thus, an attacker who moves the code to a machine can be detected by looking for this particular signature. This is, of course defeated if the attacker is smart enough to encrypt the code first. However, the code must eventually be decrypted, which gives a sufficiently paranoid operating system the opportunity to detect the signature. Such an operating system would presumably be invulnerable to this kind of attack. After all, why go to all this trouble when it is possible to make the operating system nearly invulnerable to buffer overflow attacks?

A moderately clever attacker can recode the attack easily enough, making the job of detection via signatures much harder, so other detection methods must be employed.

If the system monitors for inappropriate changes of permission, the attack can be detected by noting that permission has been changed without the appropriate legal sequence of events. This is much more reliable than the signature approach.

5.2.3.2 Race Condition Imagine that you want to change an entry in the password file. Why would you want to do this? One reason would be to change the root password from something you do not know to something you do. If you could only write to the password file, you could do this and you could then gain root permission. This has the added benefit that nobody else would have root permission anymore, since they would not know the new root password. Like Yertle the Turtle, you would be king of all you could see!

The problem with this is that you cannot write to the password file unless you already have root permission. The solution to this is to trick a program that has root permission into writing the file for you.

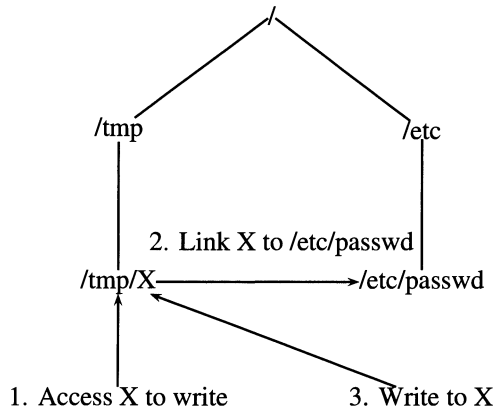

Fig. 5.2 An example of a race condition allowing write access to a password file. After opening the file /tmp/X for writing, but before writing any data, the file is deleted and replaced with a link to /etc/passwd. Subsequent writes then end up in this file.

When a program goes to open a file, it first checks (with a system call) to see whether it has permission to access the file. Then, if the answer is "yes," it opens the file (with a second system call). The problem comes in when the file changes between the two system calls. This is illustrated in Bishop and Dilger [1996]. Figure 5.2 shows the steps.

Suppose that the attacker wishes to overwrite the file /etc/passwd (the Unix password file) with the file /home/userx/mypasswd. The /tmp directory is world-writable (that is, writable by any user), so it affords a nice place to perform this attack.

1. The attacker creates a file in /tmp, say /tmp/X.

2. The attacker program checks to see whether it has permission to open /tmp/X. It does, since it is the user's file.

3. Before the program opens the file, the attacker removes /tmp/X and makes a hard link between /tmp/X and /etc/passwd.

4. The program then opens /tmp/X and copies /home/userx/mypasswd into it and hence into the /etc/passwd file.

This kind of vulnerability is referred to as a time-of-check-to-time-of-use (TOCTTOU) flaw. As in the preceding example, it comes about when a program checks an object for a property and then assumes that the property still holds when it goes to perform some operation on the object. If the property no longer holds when the second operation is performed, a security fault occurs.

A similar example is given in Ko et al. [1994]. There is a program, *rdist*, which is used to maintain file consistency across a number of hosts. When rdist updates a file, it creates a temporary file, copies the data into the file, and changes the

permissions on the temporary file to match those of the copied file using "chown" (change owner) and "chmod" (change mode). Then, it renames the file to the correct name.

Suppose an attacker (who already has obtained a local shell on the machine) wishes to change the "suid" bit on /bin/sh. Recall that /bin/sh (the Bourne shell) is owned by root, and if the suid bit is set, then the program is run as the owner of the program. Thus, if one can set the suid bit on /bin/sh, one can obtain a shell that is running as root, thus obtaining root access on the machine. Unfortunately for the attacker (but fortunately for everyone else), one cannot (normally) set the bits on a program one does not own, so some kind of trick must be employed.

The trick, using rdist, is as follows. The attacker updates a file local to the machine with the appropriate bits set. After rdist opens the temporary file and has started copying the data, the attacker renames the file and makes a symbolic link to /bin/sh with the name of the temporary file. After rdist has finished copying (note: the copy continues into the renamed file, not the new linked file), it runs chown and chmod (this time on the symbolic link file). The trick is that chown does not follow symbolic links (so /bin/sh remains owned by root) but chmod does, changing the bits on /bin/sh. The attacker then runs /bin/sh and has root.

Bishop and Dilger [1996] give several other examples of these kinds of problems. They also describe some approaches to detecting these flaws. The first involves a code checker that searches source code for potential flaws. The second is a dynamic approach that watches the run-time environment for potential TOCTTOU flaws. A third approach, not mentioned by Bishop and Dilger [1996], would be to use a file integrity checker such as tripwire (Section 5.6.6) incorporated with a monitor such as lsof (Sections 1.9.11 and 5.6.3) to see who is opening the various files. This is potentially quite expensive from a computational standpoint since the checks must be done essentially continuously. That makes this an impractical solution. The real solution is to write the operating system so that it is invulnerable to this kind of attack, which is no easy task.

A similar attack, which effectively destroys a file that the user otherwise would not have permission to touch, can be found at

www.rootshell.com

(search for gcc). The idea is that gcc, the Gnu C compiler, uses temporary files in /tmp for its intermediate files. The script watches the /tmp directory for files whose names match those used by gcc. When it finds one, it links the victim file to the file used by gcc. When gcc outputs the temporary data, it overwrites the victim file.

This attack is purely destructive since the attacker has no control over the content written to the file. It can be detected with a file integrity checker such as tripwire, provided the victim file is one of the ones protected. It can also be detected by noticing garbage that looks like the output of a C compiler in files that should have something else in them.

5.2.4 Covering Up

Once an attacker has gained access to a machine, the first order of business is usually to cover their tracks to make detection (and, ultimately, prosecution) more difficult. There are a number of techniques that are useful for this purpose.

The simplest thing to do is to hide any new files that the attacker has put on the machine by starting their name with a ".". These files are not listed in normal directory listings. One must execute a "ls -a" to see these "hidden" files. A related method is to make a directory called "...". In Unix, the directory "." is the current working directory, ".." is the previous directory, and "..." will often be overlooked.

The next thing to do is to remove any traces from audit logs. There are various tools that will help with this. It only works if the logs are accessible (which is an argument for using a log server, making the log files inaccessible on any machine eccept the log server itself). Once this is done, ps and netstat can be replaced with trojan programs that do not display the attacker's processes or connections.

5.3 NIDES

The Next-generation Intrusion Detection Expert System (NIDES) (Anderson et al. [1995]) was developed by SRI in the early 1990s (see Javitz and Valdes [1991] and Javitz and Valdes [1993]). We will concern ourselves mainly with the statistical component of the NIDES system. NIDES utilizes logfile entries to extract information about various activities such as file access and cpu usage. The idea is to construct statistics on these usages under normal conditions then use the statistics to test for abnormal usage in subsequent operation.

NIDES operates by measuring various activity levels for a set of defined activities and combining these into a single overall measure of the "normality" of activity for the recent past. This statistic, denoted T^2, is then tested against a predefined threshold to determine whether the recent activity is sufficiently "abnormal" to warrant alerting the security officer.

NIDES uses a wide range of disparate data to make its assessment, which makes it a particularly interesting approach from a statistical viewpoint. The data types are broken into four categories:

- **Intensity measures** An example would be the number of audit records generated within a set time interval. Several different time intervals are used in order to track short-, medium-, and long-term behavior.
- **Distribution measures.** The overall distribution of the various audit records is tracked via histograms. A difference measure is defined to determine how close a given short-term histogram is to "normal" behavior. These measures could properly be treated as functional data (Ramsay and Silverman [1997]).
- **Categorical data** The names of files accessed or the names of remote computers accessed are examples of categorical data used.
- **Counting measures** These are numerical values that measure such things as the number of seconds of CPU time used (to an accuracy of about a

microsecond). They are generally taken over a fixed amount of time or over a specific event, such as a single login. Thus, they are similar in character to intensity measures, although they measure a different kind of activity.

A set of measurements is defined from the preceding categories, measuring such things as CPU usage, number of files accessed, which files were accessed, and elapsed time for different applications. These measurements are used to generate a statistic denoted S, and the T^2 statistic is defined as a sum of the squares of the S_j:

$$T^2 = \frac{1}{n}\sum_{j=0}^{n} S_j^2. \tag{5.1}$$

The developers of NIDES suggest that future work look at correlations between the S_j as an area that might provide useful information. To my knowledge, this has not yet been investigated.

The NIDES approach is to compare recent performance with past performance. One way that the developers could have chosen to implement this is to use time windows on the data. One could compute a statistic on a window of, say the last ten seconds and then compare this with values taken over windows in the past to determine whether the statistic has changed.

5.3.1 Statistical Calculations in NIDES

Host-based detection should occur in real time, so computational efficiency is essential. Thus, one would implement the window approach efficiently by updating current values rather than recomputing them. For example, if one were computing an average of values $A_{t,n} = (x_{t+1} + x_{t+2} + ... + x_{t+n})/n$, one would compute $A_{t+1,n}$ as

$$A_{t+1,n} = A_{t,n} - x_{t+1}/n + x_{t+1+n}/n. \tag{5.2}$$

Similar "downdate/update" strategies are available for other statistics that one might wish to compute on these windows.

A related formulation is the recursive update formula for the mean and variance of a random variable. They are introduced here so that they can be used as a simple procedure for implementing exponential windows. The formulas are shown in Equations (5.3)–(5.5).

$$\bar{x}_{n+1} = \bar{x}_n + \frac{1}{n+1}(x_{n+1} - \bar{x}_n), \tag{5.3}$$

$$\hat{S}_{n+1} = \hat{S}_n + \frac{n}{n+1}, (x_n - \bar{x}_n)'(x_n - \bar{x}_n) \tag{5.4}$$

$$\hat{\Sigma}_{n+1} = \frac{1}{n}\hat{S}_{n+1}. \tag{5.5}$$

These are easy to derive. For example, for $x_1, \ldots, x_n$, the sample mean calculation is

$$\overline{x}_n = \frac{1}{n}\sum_{i=1}^{n} x_i. \tag{5.6}$$

Now, assume that we obtain a new observation x_{n+1}. We can recompute the sample mean via Equation (5.6), but we'd like to simply update the mean we already have.

$$\begin{aligned}
\overline{x}_{n+1} &= \frac{1}{n+1}\sum_{i=1}^{n+1} x_i \\
&= \frac{1}{n+1}\sum_{i=1}^{n} x_i + \frac{1}{n+1}x_{n+1} \\
&= \frac{n}{n+1}\overline{x}_n + \frac{1}{n+1}x_{n+1} \\
&= \overline{x}_n - \frac{1}{n+1}\overline{x}_n + \frac{1}{n+1}x_{n+1} \\
&= \overline{x}_n + \frac{1}{n+1}(x_{n+1} - \overline{x}_n).
\end{aligned} \tag{5.7}$$

A similar calculation can be used to derive Equation (5.5).

It should be noted that the order of the updates of the recursive formulas is important. As seen in Equation (5.4), the update of S_n uses $\overline{x}_n$, not $\overline{x}_{n+1}$, so code to implement these equations should update S before updating the mean.

The NIDES development team chose to take a slightly different approach. They put an exponential window on the observations rather than a rectangular one. In this way, the current statistic depends not only on a small window in time but on all the data for all time (in principle, although in practice since the dependence on past data drops off exponentially there is little dependence on data a few half-lives in the past). An exponential window can be implemented in the preceding recursive formulas by setting the ns on the right-hand side of the equalities in Equations (5.3) and (5.4) to some fixed value, say N.

Each S statistic is computed from a "raw" statistic denoted Q. We will first investigate the kinds of measurements that might correspond to Q statistics and consider how the Q statistics are computed. We will then describe how an S statistic is computed from a given Q for the different data types.

5.3.1.1 Intensity Measures Intensity measures are counts of audit records per fixed time unit. The idea is to get a measure of the overall activity level of the system (or the user if the records are restricted to those generated by the user's processes). In its simplest form it is a simple count of the number of audit records, however one could easily implement different counts for different types of records.

We will deviate from the notation of the NIDES report (Anderson et al. [1995]) at this point. Rather than referring to all measures as Q, we will denote intensity measures as I. Thus, the intensity at time t is denoted I_t, and the intensity after

n records is I_n. One first initializes the intensity value to I_0, some initial value that is chosen to be a reasonable start. Generally, one either sets $I_0 = 0$ or sets it to some value determined by considering average values for a number of related data sets. The formula for updating I_n from the $(n+1)$st audit record is

$$I_{n+1} = 1 + 2^{-r\Delta t} I_n, \tag{5.8}$$

where Δt is the time between the nth and $(n+1)$st audit records and r is the decay rate, determining the rate of decay of the exponential window. In this case, and throughout the NIDES discussion, the rate is discussed in terms of half-life, for obvious reasons. This formula is recursive since it only requires the previous value of the statistic and the elapsed time in order to update the value.

The NIDES implementation discussed in Anderson et al. [1995] implements three intensity measures with half-lives of 1, 10, and 60 minutes.

5.3.1.2 Audit Record Distribution Measures The first thing that comes to mind upon looking at the intensity measures is the question of the distribution of the audit records. The intensity measures are in effect averages of the time between records, taken over different scales. An obvious question is how audit records are distributed. Audit records describe different types of behavior, such as file access, I/O, network access, and so on. Audit record distribution measures try to take into account the overall distribution of the different types of activities.

First, one determines the different activity types that will be monitored. This can be done by reviewing the log files for a period of time to determine what things are typically logged and by reading the documentation for the different logging programs to determine what kinds of activity are typically logged. This will be different for different architectures; however, there are a number of things (such as those mentioned previously) that will be pretty much universal.

Once a set of activity types has been defined, NIDES computes the relative frequency of occurrence of each type and compares this with historical (longer-term) values for these types. This amounts to computing the weighted sum of squared differences between the observed and historical rates, weighted by a measure of the variance of the historical estimates. In order to be precise, we need to define some of the values used in the calculation.

The sample size for the statistic, N_r, is defined as the sum of the decay weights:

$$N_r = \sum_{j=1}^{n} 2^{-r(n-j)}. \tag{5.9}$$

The audit record distributions concern daily tabulations of audit records of different types. Let $W_{j,m}$ be the number of audit records of type m that were observed on day j, and W_j the number of audit records of all types on day j. Just like with intensity measures, we take weighted averages to allow a sliding window in the calculation, so define

$$N_k = \sum_{j=1}^{k} W_j 2^{-b(n-j)}, \tag{5.10}$$

the exponentially weighted total number of records that have occurred. In this case, the decay rate has been denoted b to indicate that it can be a different value than the one used in the calculation of I_n. The NIDES report notes that a recursive calculation of N_k can be defined as

$$N_k = 2^{-b}N_{k-1} + W_k. \tag{5.11}$$

Let $g_{m,n}$ denote the short-term relative frequency for activity type m, computed upon the observation of the nth record. This is computed as

$$g_{m,n} = \frac{1}{N_r}\sum_{j=1} n2^{-r(n-j)}I(j,m) \tag{5.12}$$

or recursively as

$$g_{m,n} = 2^{-r}g_{m,n-1} + I(n,m)/N_r, \tag{5.13}$$

where $I(j,m)$ is the indicator function indicating whether the jth record was of type m.

Note that $g_{m,n}$ is updated for every audit record whether it is of the appropriate activity type or not. This puts some constraints on the number of activity types that are practical to monitor. For example, it is probably not feasible to consider every type of TCP connection (each possible port) as being a different activity type since this requires the updating of 65536 values for every record. Although this can easily be done on modern systems, the computational overhead of this is probably more than can be justified.

Letting $f_{m,n}$ denote the long-term historical frequency of occurrence of activity type m as of record n, we compute

$$f_{m,n} = \frac{1}{N_r}\sum_{j=1} n2^{-b(n-j)}W_{m,j} \tag{5.14}$$

and

$$V_{m,n} = \min(0.01, f_{m,n}(1 - f_{m,n}))/N_r. \tag{5.15}$$

The statistic, which we will denote D_n, for the distribution of the audit records is then defined as

$$D_n = \sum_{m=1}^{M}(g_{m,n} - f_{m,n})^2/V_m. \tag{5.16}$$

5.3.1.3 Categorical Measures Categorical measures are computed exactly like distributional measures except that only the bin associated with the categorical value is updated. Thus, if we modify Equation (5.13) as in Equation (5.17) below, we can compute the statistic for categorical values in essentially the same manner as before:

$$g'_{m,n} = 2^{-r}g'_{m,n-1} + 1/N_r, \tag{5.17}$$

$$C_n = \sum_{m=1}^{M}(g'_{m,n} - f_{m,n})^2/V_m, \tag{5.18}$$

where, of course, all the values are calculated for the categorical variables.

5.3.1.4 Counting Measures Counting measures are computed by converting them to categorical variables. Recall that counting measures are things such as CPU usage, which are naturally measured in terms of counts, such as number of milliseconds of CPU usage. These are binned into 32 ranges, which are then treated as categorical measures and treated as before.

5.3.1.5 Computing S from Q Since the Q values defined earlier are from quite disparate distributions, some kind of normalization is needed to allow the simple combination defined in Equation (5.1) to make sense. This is accomplished as follows. For each intensity measurement, I_n, a histogram is made of historical values; that is, one considers several time periods in which the I_n were computed, and constructs a histogram of the values observed. This histogram typically has 32 bins, with the last bin consisting of all instances of I_n above the lower bound for the bin. The histogram then is an estimate of the density of the I_n, and this in turn determines an estimate of the distribution function as the sum of bin frequencies for bins with ranges less than or equal to the observed value I_m. An obvious modification to this would be to use the empirical distribution function F for the historical data; however, this requires retention of all the data, and this approach was not taken, presumably for reasons of computational efficiency. The value of S for the intensity measure is then

$$S_n = \Phi^{-1}(1 - F(I_n)/2), \tag{5.19}$$

where Φ is the distribution function for the standard normal.

This histogram is defined in a manner similar to that used for the computation of the Q measures themselves. For bin m, the value of relative frequency with which Q is in this bin is

$$F_{m,n} = \frac{1}{N_n} \sum_{j=1}^{m} W_{m,j} 2^{-b(n-j)}, \tag{5.20}$$

where in this case the $W_{m,j}$ is the number of audit records on day j that fell into bin m.

The nonintensity measures are treated similarly, using Equations (5.20) and (5.19), except that the sum of bins greater than or equal to the observation is used.

5.3.1.6 New and Rare Categories NIDES allows the creation of new categories. Because of the decay discussed earlier, these may actually be old categories that have not been used recently, or they may be genuinely new.

The mechanism for handling novel events is a separate category labeled "new." When a novel event is observed for the first time during a day, the "new" category's short-term probability is incremented. The short-term probability is then compared to the long-term probability to determine whether there is a significant amount of "new" activity. If there is, a new category can be created and/or the system administrator can be notified.

Similarly, the rarest categories are aggregated into a single "rare" category. In this manner, while an intruder that touches a few rare categories may not be detected for the individual categories, the large amount of activity in the "rare" category may be enough to signal a problem.

5.3.1.7 T^2 and Alerts Once a T^2 has been observed, a decision must be made as to whether this value is large enough to indicate abnormal behavior. There are several points to consider. First, the value of T^2 at audit record n is highly correlated with that for record $n + 1$. This means that some kind of memory of past alerts needs to be kept to avoid having the system give redundant alerts once T^2 goes over the alert threshold the first time. This dependence makes the analysis of the statistic rather difficult, although it is approximately χ^2, being the sum of squares of values which themselves come from sums of random variables. Rather than attempt any kind of theoretical analysis of this statistic, however, it is probably sufficient to set the threshold via an empirical study. For example, one could collect the statistic for a period of time in which one knew when the activity was "normal" and when attacks had been mounted and use these data to set the threshold. This once again brings up the issue of how one gets nice clean data such as this, which was addressed to some extent in Chapter 3.

5.3.2 NIDES Performance

The performance of NIDES in several experiments is reported in Anderson et al. [1995]. NIDES has several parameters (most notably the half-life) and a threshold to adjust. The authors performed a set of experiments with varying values of the parameters to determine the possible range of performance measures.

Thirty programs were identified as having sufficient examples within a data set that had been collected. The activity of these programs was then monitored, and NIDES was evaluated to determine the false alarm rates on these data. These rates varied between programs and the parameters that were set. In one experiment, NIDES performed at false alarm rates between 0 and 13% for the individual programs. For example, the best performance reported in this experiment was that four of the programs had false alarm rates larger than 1%. Thus, the performance was quite dependent on the activity of the program, presumably on the variability of "normal" activity for that program. Other experiments showed results in the range from 0 to 5% false alarms.

To determine the detection probabilities, other programs were substituted (and renamed) in place of the "normal" programs, and the task of NIDES was to determine that a substitution occurred. For the most part, it was able to do this, although not always in all the places where a substitution occurred. The probability of detection ranges (depending on experiment run and application programs considered) from around 72% to 100%. The latter is only attained for specific programs, not for the overall problem of detecting the masqueraders within all the data.

Overall, these results are not bad, considering the difficulty of the task attempted. We will see other approaches to this and related problems in the next section and in Section 5.5.2.

5.4 COMPUTER IMMUNOLOGY

Consider the problem of determining whether a program is operating normally or whether something has gone wrong. If the program is acting in an unusual manner, it is possible that it has been compromised by an outside agent, perhaps resulting in system compromise. How can we determine whether the program is operating normally?

If the program is simple and well-understood, one can simply list all the possible "normal" actions that the program could take and check this list in subsequent program evaluations. However, for most non-trivial programs, this is a daunting task.

One way to characterize the actions taken by a program is to list the sequence of system calls it makes. In order to effect any changes to the system, the program must use system calls to interact with the system (for example, to store something in a file), so considering the sequence of system calls is a reasonable place to start.

Unfortunately, there are quite a number of possible system calls, and the sequence of calls performed is data-dependent, so one must perform some kind of statistical analysis to determine what "normal" sequences look like. This is done by considering strings of system calls of a fixed size n, so-called "n-grams" (see, for example, Forrest et al. [1996]) . First, define an alphabet of symbols which correspond to all system calls. Fix a length n, which will be the length of characteristic strings. Then, for many "normal" operations of the program in question, keep a list of all "n-grams" observed for that program.

Note that these "n-grams" are not independent. Consider the following example:

A B B A C D D A E A B B C A E D

This results in the following set of 7-grams:

A B B A C D D
B B A C D D A
B A C D D A E
A C D D A E A
C D D A E A B
D D A E A B B
D A E A B B C
A E A B B C A
E A B B C A E
A B B C A E D

In this alphabet consisting of five symbols, there are $5^7 = 78,125$ possible distinct 7-grams. One question of interest is how long one has to observe the program in order to be confident that one has seen a given percentage of the "normal" n-grams for the program.

On a Linux machine, the command to trace the system calls of a program is strace. For example, the following is a shell script to trace any command:

```
#!/bin/csh -f
strace -o $1.$$ $*
```

If the preceding is in a file called "mytrace," then running

```
mytrace vi main.c
```

results in a file named vi.1986 (assuming that when mytrace was run it was given the process ID 1986) containing a listing of all of the system calls made by the vi process.

I ran a small experiment to illustrate the process. Consider the problem of characterizing the operations of the GNU C-compiler, gcc, in this manner. First, I compiled the R language (Ihaka and Gentleman [1996] on a Red Hat Linux 6.1 machine. R is a language very similar to S, a statistical language. R is public domain software, with both executable and source distributions. The version I used (0.90) made 446 calls to gcc. There were 19 distinct system calls, listed in Table 5.1.

With $n = 7$, there were 122 distinct n-grams throughout the 446 calls to gcc. Is this representative of the "normal" activity of gcc? The number of n-grams as a function of the number of files read is plotted in Figure 5.3. Note the relatively long period where no new n-grams are defined, followed by a sharp increase in the number of n-grams. This increase at the end is evidence that we have not yet found all the n-grams for gcc.

A much more extensive experiment is shown in Figure 5.4. A wide range of programs were compiled, and the unique 7-grams were tallied at the end of each file. Figure 5.4 shows the total number of unique 7-grams plotted against the number of files processed. Two new system calls, "pipe" and "write," were added for a total of 21. Thus, there are 21^7, or nearly 2 billion, possible 7-grams.

It appears clear from the curve in Figure 5.4 that we have not yet completely characterized the normal behavior of the program gcc. We would therefore need to collect more data on the operations of gcc. As an anecdote, after compiling a large number of programs, I compiled everyone's first C program, "hello world,"

Table 5.1 System calls made by gcc during compilation of R.

access	brk	close
execve	_exit	fstat
getpid	gettimeofday	mmap
mprotect	munmap	open
personality	read	rt_sigaction
stat	unlink	vfork
wait4		

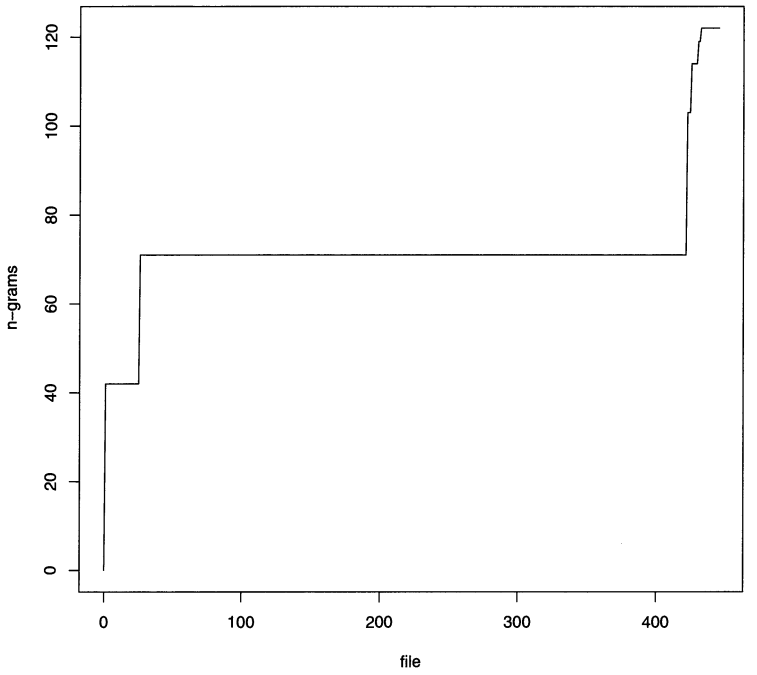

Fig. 5.3 gcc n-grams.

and discovered a new 7-gram. Clearly, more data are needed. The question is how much more data are needed?

One way to approach answering this question is to consider the work on estimating the probability of discovering a new species, for example Starr [1979], Chao [1981], and Bickel and Yahav [1986]. See also Finch et al. [1989]. Another approach would be to model the system calls as a Markov process and then run the model for a very long time. This is the approach of Dan Naiman of The Johns Hopkins University. (This work is unpublished, but was presented at the 2000 Souther Regional Council on Statistics Summer Research Conference in Statistics). He found that a simple Markov model leads one to the conclusion that we have not come close to finding all the "normal" n-grams for gcc.

The seminal work on using n-grams for intrusion detection is by Stephanie Forrest and her team at the University of New Mexico. It is documented in several papers, such as Forrest et al. [1994], Forrest et al. [1996], Hofmeyr et al. [1998], and Warrender et al. [1999]. The idea behind the n-gram approach is that any attack (such as a buffer overflow attack) that compromises a particular program will cause that program to either execute system calls that it does not normally execute or execute system calls in an unusual order. By characterizing the normal pattern

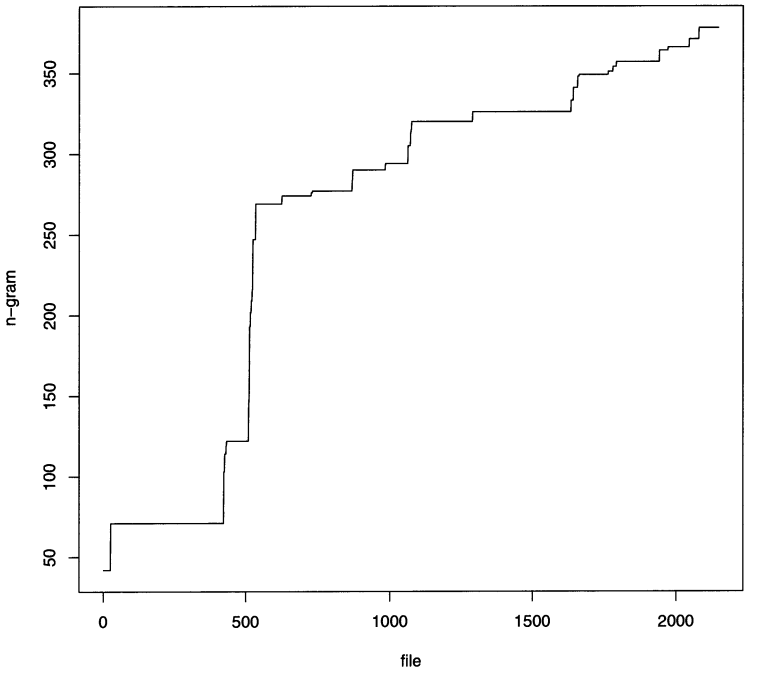

Fig. 5.4 A more extensive experiment with gcc n-grams.

of system calls, one can detect attacks by detecting never-before seen patterns of calls.

The n-gram approach proceeds as follows. Define the size, n. The authors suggest 6 as a reasonable value. Run the program many times, storing the unique strings of size n observed in a database.

As seen in the preceding experiment, care must be taken in the construction of the database. One way to populate it would be to monitor the program during normal operation. A program like gcc, which may be used often on some machines and rarely on others, is particularly problematic. As the experiment shows, the data (the C programs that gcc compiles) have a large impact on the order of system calls, so one needs to ensure that the test period is a proper sample of the program execution.

Now define a window size W. This is suggested to be 20 in most of the work cited earlier. For a new observation of the program execution, tally the number of sequences within each window of size W that do not match any sequence in the database. If this number is larger than some threshold, signal an alert. An alternative to this measure would be to allow partial matches of the sequences and score the mismatch according to the Hamming distance. This is described in Hofmeyr et al. [1998].

The first paper, Forrest et al. [1994], takes a slightly different approach, which we will see again in Section 6.5. The idea is to construct a collection of "detectors" aimed at detecting things that have never been seen. This is intended more to protect a particular piece of code from being changed rather than for the detection of unusual behavior, but it could be used for the latter as well.

Forrest et al. [1996] describe the n-gram idea as applied to the sendmail program. They found that after approximately 10,000 system calls, they had obtained a database of roughly 1400 6-grams for sendmail. Further experimentation showed that attacks against sendmail produced a significant number of 6-grams that did not appear in their "normal" database, thus indicating that the basic idea is sound.

Warrender et al. [1999] evaluate several methods for detecting intrusions via n-grams. They used data on several different programs, such as lpr, xlock, and login, and attacked each application with an exploit designed for that application. This gives them "normal" data as well as intrusions. They collected data for their models until the rate of increase in the number of new sequences dropped below a preset value. As we have seen in our small experiment (Figure 5.4), this curve can be quite rough, with long flat periods and sudden jumps. Hence, the authors smooth these curves prior to calculating the rate of increase.

Several techniques were compared, including the "standard" n-gram approach described earlier, a version where the relative frequencies of the "normal" n-grams were used, a rule learning system called RIPPER (Repeated Incremental Pruning to Produce Error Reduction - an algorithm from the machine-learning community), developed by William Cohen (Cohen [1995]), and a hidden Markov model (HMM). The results of the study show that the HMM did quite well, as did the standard approach. No method was universally superior, and there was some evidence that there was insufficient training data for the more complicated models. Still, the results were promising, particularly the fact that the simplest model performed quite well. The bottom line was a probability of detection in the high 90% range with a false alarm rate on the order of 1/10,000–1/1000 for the best algorithms. These results are, of course, preliminary, due to the size of the training and test sets and the small number of attacks available for testing, but, as mentioned in Chapter 3, all researchers must struggle with this is a problem.

A related approach is described in Hofmeyr and Forrest [1999], Hofmeyr and Forrest [2000], and Forrest and Hofmeyr [In press]. These papers describe a technique for developing an "immune system" for computers (or networks) with detectors that look for "self" and "nonself." This work will be discussed in more detail in Section 6.5. See also Somayaji et al. [1997] for some thoughts on immune systems for computers.

In Somayaji and Forrest [2000], a technique is described for responding to intrusions. The idea is that once anomalies are detected, the monitor can abort or delay system calls, thus stopping the attack. This integrates the response into the "immune system."

5.5 USER PROFILING

Most systems have a method of user authentication. This usually consists of a required user name and password. Once this information is obtained, however, the attacker is free to access the system just as the legitimate user would. There are several ways one could go about stopping this. In this section, we will look at attempts to detect when the user is not the person authorized to use the account.

The basic idea behind these user profiling methods is to measure something about the way the user interacts with the computer and use this to determine a profile of the user. These measurements may be biometric, such as keystroke timings, or even finger or palm prints or retina scans, or they may be measures of activity, such as which commands are executed and in what order. If the person accessing the account does not match the profile, the assumption is that the person is an attacker. For a discussion of some of the biometric techniques available, see Miller [1994]. Although this paper is somewhat dated, it is written at an accessible level.

Several possible measurements can be made, but we will concern ourselves only with those that can be made on any computer, without specialized hardware such as palm readers or retina scanners. A short list (mostly taken from Shepherd [1995]) includes:

- **Intervals between keystrokes** For example, when the user types a password, the computer retains the timings between the different characters, and compares these against stored patterns. Similarly, one could measure how long the key is depressed and, on some systems, the force of the keystroke.
- **Mistypings** People tend to make the same typing mistakes over and over.
- **Typing speed** This is obviously context-dependent (text versus programming, for example).
- **Text or command statistics** People tend to use the same commands over and over (for example, most people, after typing cd to move to a new directory, will type ls to see what is there. Some of us type the ls command even when we know what is there or don't really care. Similarly, people have words or phrases they use habitually. For example, I have a colleague who likes to use "heretofore".
- **Mouse events** How often one uses the mouse, how fast the mouse moves, the timings of mouse events are all potentially useful for constructing user profiles.
- **Computer usage statistics** Examples would be the amount of memory or CPU usage, which disks/directories are accessed, or whether the access is from the console, telnet, rlogin, or ssh. Another potentially interesting statistic is the number/type of system calls resulting from the user's actions.

Several issues need to be considered in developing a user authentication technique based on a profile such as those just listed. These are not unique to the

problem of user authentication, and in fact are relevant to most intruder detection problems:

- What data need be collected?
- Does the user select the word/phrase to be measured? For example, passwords tend to be selected by the user, and are often easy to type (partly due to practice). Alternatively, one could collect data for several words or phrases and present a randomly selected one for authentication purposes. This makes it more difficult for an attacker to mimic the authorized user.
- What method of classification is to be used?
- Is it to be used once, for example in conjunction with a user name/password, or will it be continuous?
- How many users need to be profiled? For example, a personal computer may only have one authorized user, whereas a main server may have hundreds.
- How much processing (CPU time) is acceptable for making a decision. For example, a system with hundreds of authorized users may require much faster authentication, in order to provide an acceptable level of service to the users, than a single-user system.
- What level of false alarms is acceptable?
- How secure does the system need to be? This is another way of asking the question of how many missed attacks are acceptable.

5.5.1 Keystroke Timings

We will consider keystroke timings first. A typical scenario is to collect the time between keystrokes within the password (with or without the final carriage return) and use this to classify the user as authorized or unauthorized.

Another possible application would be online authentication. In this application, the machine would monitor the activity of a user - for example, the timings of keystrokes for commonly used words - and try to determine whether a masquerader is at work. This could either be done by assuming the user is legitimate and looking for a sufficient deviation from normal or by requiring the user to remain in the "normal" range in order to remain online.

The data are a vector of timings, one for each adjacent pair of characters in the password. Since different people use different passwords, this means that the problem really reduces to one of a true hypothesis test: the null hypothesis being that the user is who they are purporting to be.

Several methods for classification are possible. We will discuss the problem as one of deciding which of a number of possible users is the one actually typing. The obvious extension to this is to classify users as "unauthorized" if they are sufficiently different from the user they are attempting to impersonate. Looking at the problem as a multiclass one allows us to consider the possibility that we may

be able to determine who the user truly is in those situations where the physical security restricts the possibilities.

A typical experiment consists of collecting samples for each user, denoted "training samples" which are used to construct a classifier. Further samples are then used to test the system to determine the performance of the classifier. Generally, one is interested in the probability that the classifier will reject a valid user (type I error) and the probability that it will incorrectly pass an unauthorized user (type II error).

The simplest classifier that has been used for this set of problems is the minimum distance classifier. The idea is to compute the mean (or median) for the training samples. The classifier then involves computation of the distance to the mean (or median), and a threshold determines whether the observation is to be classed according to its closest class or rejected as unknown. The distance is generally taken to be Euclidean,

$$d_i^2(x) = (x - \mu_i)^t (x - \mu_i), \tag{5.21}$$

where μ_i corresponds to the mean for class (user) i. In the case of the median, the same equation is used with median in place of mean.

The minimum distance classifier is a type of linear classifier. An obvious extension of this idea is to use the covariance matrix as well as the mean, producing a quadratic classifier,

$$d_i^2(x) = (x - \mu_i)^t \Sigma^{-1} (x - \mu_i). \tag{5.22}$$

Another commonly used classifier is the k-nearest neighbor classifier. First, consider the nearest-neighbor classifier (see page 77). Given a training set and a new observation, classify the observation according to the class of the closest training observation. The k-nearest neighbor classifier extends this idea by in effect voting amongst the k closest training observations.

Bleha and Gillespie [1998] compared three simple techniques for classification of keystroke data. The first was a simple minimum distance classifier. The two others involved extracting features from the timing vectors (for example, averaging the first three times, the next three times, and so on, to produce a vector one-third as long as the original). Their conclusion was that the best technique was to use the original data and the minimum distance classifier. This was not an extensive examination of possible classifiers but rather compared two specific feature extraction methods and determined that they were not preferred over the original data.

There have been many papers written about the use of keystrokes for user authentication (for example, Bleha et al. [1990], Bleha and Obaidat [1991], Brown and Rogers [1994], Lin [1997], Obaidat and Sadoun [1997], and Maisuria et al. [1999]). These use the minimum distance classifier described earlier, quadratic classifiers, nearest-neighbor classifiers and neural networks, to classify keystroke timing vectors by user.

For example, in Bleha et al. [1990] and Bleha and Obaidat [1991], several experiments are described in which the linear and quadratic classifiers are compared to each other and to classifiers constructed on features, such as Fisher's linear

discriminant (FLD). In these experiments, each user typed the same phrase (UNIVERSITY OF MISSOURI COLUMBIA) and the task was to distinguish among the users.

The Fisher linear discriminant attempts to find the projection of the data that provides the best separation between the classes. If we define the scatter matrix for class i as

$$S_i = \sum_{x \in C_i} (x - \mu_i)(x - \mu_i)^t, \tag{5.23}$$

where C_i contains the data from class i and μ_i is the sample mean of class i then the FLD projection is

$$w = (S_1 + S_2)^{-1}(\mu_1 - \mu_2). \tag{5.24}$$

See Duda et al. [2000], or any other book on pattern recognition for the details of the derivation.

The results reported on the preceding experiments were on the order of 3% type I error and 0.5% type II error (Bleha et al. [1990]) or a total misclassification error rate of about 1% (Bleha and Obaidat [1991]). These studies involved ten users studied over several weeks.

Obaidat and Sadoun [1997] performed similar tests with 15 users and included several neural network classifiers, with comparable results. With thresholds set to detect all unauthorized users, Brown and Rogers [1994] report false alarm rates between 14% and 40% for the minimum distance classifier and two neural networks. Lin [1997] reports performance slightly better than the results discussed above.

Several issues are ignored in most of the papers in the literature. For example, typing errors are usually eliminated from the data. In a password system, this seems to be a reasonable approach (after all, a mistyped password is a failed authentication, which keystroke timings should never override). In most password authentication systems, however, the user is allowed to correct mistypings (using the backspace). In this case, the interkeystroke timings are changed dramatically, causing many false rejections. Most systems do not take this into account. Similarly, for online authentication, mistypings are both a source of data and a complication that must be addressed. Robinson et al. [1998] report typical password mistyping errors of over ten%.

Figure 5.5 shows some timing data for three users. Each user typed the 10 words:

home mark dart start hello dash fast task past mask

in order. The time between keystrokes within words was measured for a total of 42 timings. Each user repeated this 100 times. The data image in Figure 5.5 shows that the three users are quite distinct overall but that some words show bigger differences than others, indicating that the choice of word to use in the authentication is important.

Table 5.2 presents the results of a nearest-neighbor classifier on the keystroke data. Recall that the nearest neighbor classifier assigns to each new observation

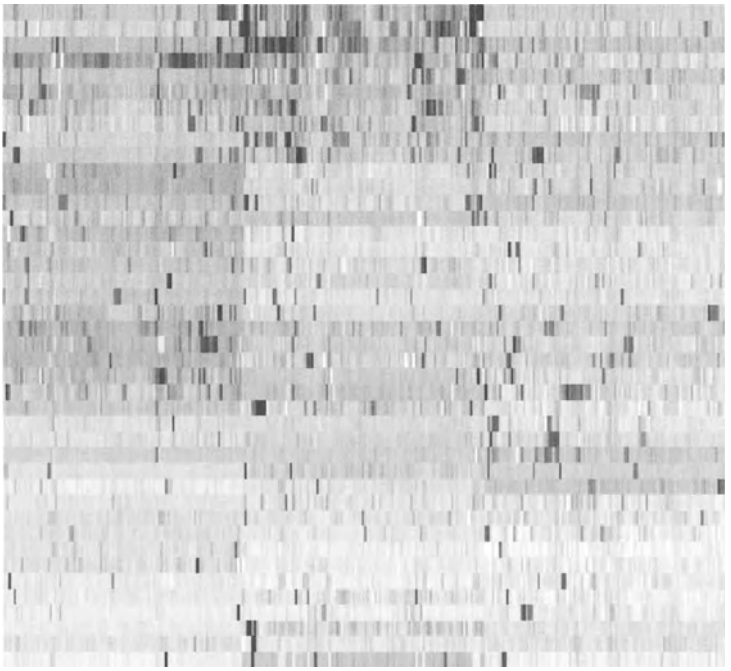

Fig. 5.5 Keystroke timings for three users. Each user typed ten words (42 characters) 100 times. Each observation (the y-axis) is then of length 42, and there are a total of 300 observations (the x-axis). The three bands visible in the image correspond to the three users.

the class associated with the observation from the training data that is closest. The results are a leave one out cross validation (see Section 3.2). Similar results are obtained using k-nearest neighbor classifiers for other choices of k. The k-nearest neighbor classifier takes a vote among the k closest training observations, classifying the new observation according to the consensus from these.

The results in Table 5.2 are in the form of a confusion matrix. This provides the information on how many observations were correctly classified (the diagonal) and which users the classifier confused. For example, the row labeled "D" shows the number of observations from user "D" that were classified as each of the users.

Table 5.2 Nearest-neighbor classifier results (confusion matrix) for the keystroke data.

	D	J	T
D	96	1	3
J	7	85	8
T	2	2	96

This indicates that of the four observations misclassified, most (3) were called user "T."

The nearest-neighbor results indicate that the users can be distinguished quite well by considering their keystroke timings for a short list of words. The performance is not perfect, however, resulting in an error of slightly less than 8%. This is comparable to the results reported in other studies, with much larger amounts of data. From a practical standpoint, this level of error is probably much too large to be of much utility.

5.5.2 Command Usage

The results on the keystroke data indicate that something other than keystroke timings needs to be used to detect unauthorized users. Although keystroke timings have their place, they are not sufficient on their own. The preceding results indicate that perhaps by monitoring typing over a long period of time, computing statistics on the timings of many words, one could build a strong authentication system based on timings of multiple words, but this still needs to be demonstrated.

Another approach would be to look at the commands that a user executes. Users will have a preference toward a certain subset of the available commands and will tend to use them in a particular order. For example, I almost always type "ls" after typing "cd," almost without conscious thought. These types of patterns may be useful for modeling user behavior and detecting unauthorized users. This idea is investigated in Schonlau et al. [1999].

Another paper that looks at this problem is Lane and Brodley [1999]. They take a machine-learning approach to analyzing the patterns of user commands. Basically, they define a similarity measure to use in comparing two sequences and define an anomaly detector which indicates when a sequence is "too unlike" a training observation for the user.

In this section, we will focus on the work in Schonlau et al. [1999]. The authors collected data on 70 users, where the command name used is recorded for each Unix command. They retained the first 15,000 commands for each user. Fifty users were denoted the "authorized" users, while the remaining 20 users were "masqueraders." Command sequences from masqueraders were then interspersed within the authorized users at random. The task was to detect the masqueraders. In order to construct algorithms to detect masqueraders, the first 5000 commands of each user were kept inviolate, so that masqueraders appeared only in the last 10,000 commands. The data were decomposed into blocks of 100 commands, and a block is either uncontaminated or it is entirely from a masquerader.

Six methods for detecting masqueraders were explored. Before discussing these, let's look at the data. Figure 5.6 (provided by the authors) shows a data image of the commands executed by each user. The commands are sorted by popularity, with the most popular commands at the top. The x-axis corresponds to the user.

Two other views of these data are presented in Figures 5.7 and 5.8. The first shows a data image (see Section 4.5.2.2) of the first 500 program calls for each user. Each column corresponds to a user, while the rows are time, with the commands numbered (alphabetically), resulting in a gray scale value for each command. The

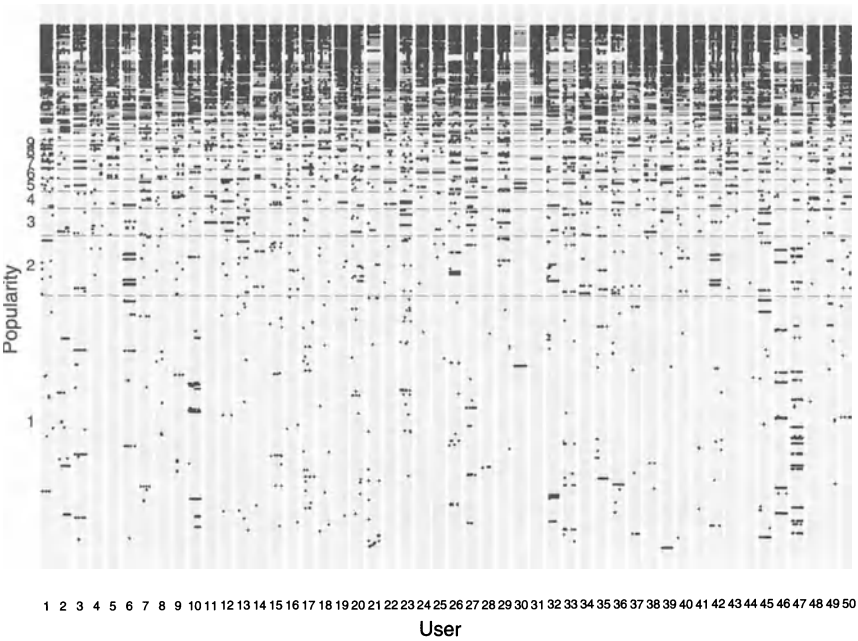

Fig. 5.6 A plot of the commands executed by each user with the commands ordered by popularity (courtesy of Matt Schonlau).

data image has been sorted by user, with the time axis left unsorted. Above the data image I have plotted a dendogram showing the user clusters used to order the plot.

There are several clusters of users evident in this plot (I count seven, but as mentioned elsewhere, selecting the number of clusters can be a matter of taste). We get a slightly different answer when we look at the data image of the interpoint distance matrix (Figure 5.8). There still appears to be some structure to these data; that is, there are individuals who appear similar.

The data images were produced using a naive choice of distance. Each program was given a number, and the Euclidean distance between vectors was used. This makes the distance between two programs an artifact of the numbering scheme. Thus, one should not make much of the clustering "discovered" in the two images. These plots do provide some interesting information, however, particularly Figure 5.7. We can see that there is quite a bit of variability within users as well as a fair amount of repetition within users.

In particular, consider the user who appears in position 42 from the left (this is actually user 30 in the original ordering). This user executes the three commands rdistd, tcsh, and rshd, repeatedly with very little deviation from this pattern. In fact, since rshd is a remote shell, the user is probably only using this system as a terminal to access a remote machine.

The Euclidean distance is not a very appropriate metric for these data. A better distance to use in this application would be simply to note whether the program used was the same or different. This is illustrated in Figure 5.9, where this distance

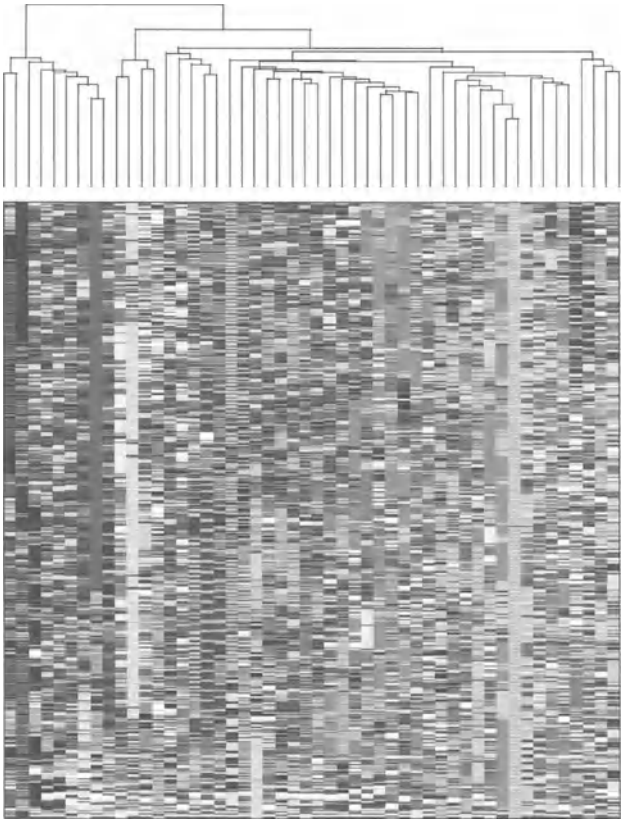

Fig. 5.7 A data image of the first 500 commands for each user. The x-axis corresponds to the users, while the y-axis corresponds to time. Only the users have been ordered.

is used instead of the Euclidean distance. In this case, we have plotted the log of the distances to enhance the plot, which otherwise is too uniform to distinguish any differences. Now, we see that the users are much more uniform, with little obvious cluster structure. This is actually the desired result since we want our users to be as different as possible in order to be able to tell them apart or, in this case, to tell them from the masqueraders.

There are many other distance metrics that one might want to use on these data. Obviously, the temporal nature of the command sequences is of some importance and so should be taken into account. However, for the most part, this structure is flexible in the sense that users can vary the pattern quite a bit and do so in the normal course of their work. For instance, users execute an occasional new command or vary the order of some commands slightly. The computational biologists have studied various metrics used for comparing DNA strands, which might be of use in this problem. Two references that cover these and related topics are Watterman [1995] and Durbin et al. [1999].

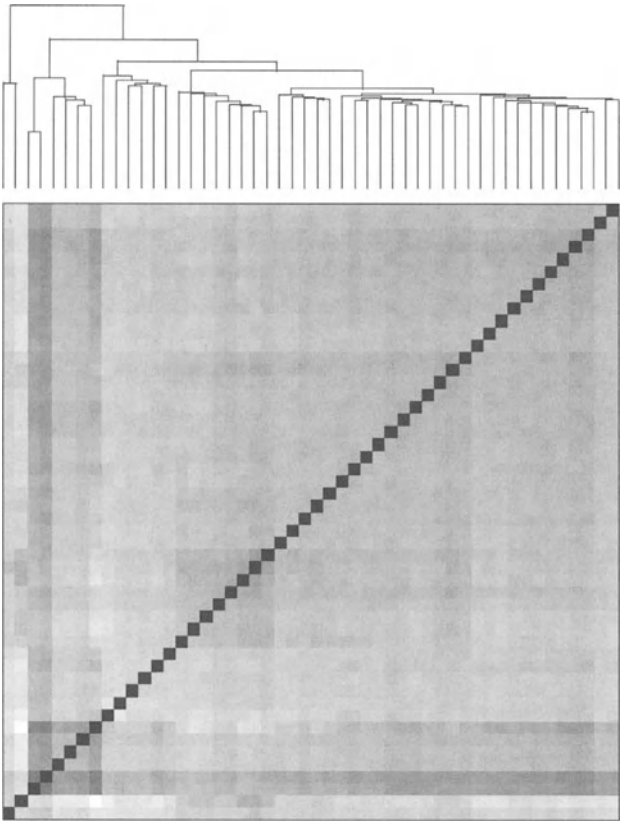

Fig. 5.8 Data image of the interpoint distance matrix for the data in Figure 5.7. In this case, the Euclidean distance between activity vectors was used.

A nontemporal measure, which simply counts the number of times a particular program is used by each user, is

$$\text{dist}(u_1, u_2) = \sum_{k=1}^{M} \frac{\|\#(u_1 = k) - \#(u_2 = k)\|}{n_k}, \tag{5.25}$$

where M is the number of programs, $\#(u_i = k)$ is the number of times user i used program k, and n_k is the number of times program k appears in the data. The data image using this metric is shown in Figure 5.10.

The original temporal data is displayed in Figure 5.11, using the ordering defined by Equation 5.25. This shows clearly that the distance ignores the temporal nature of the data since there is no obvious correspondence between the clusters and the vectors of the data.

We now turn to the question of whether there is any structure to the data for an individual user. For this, we take the first 5000 commands for each user and break

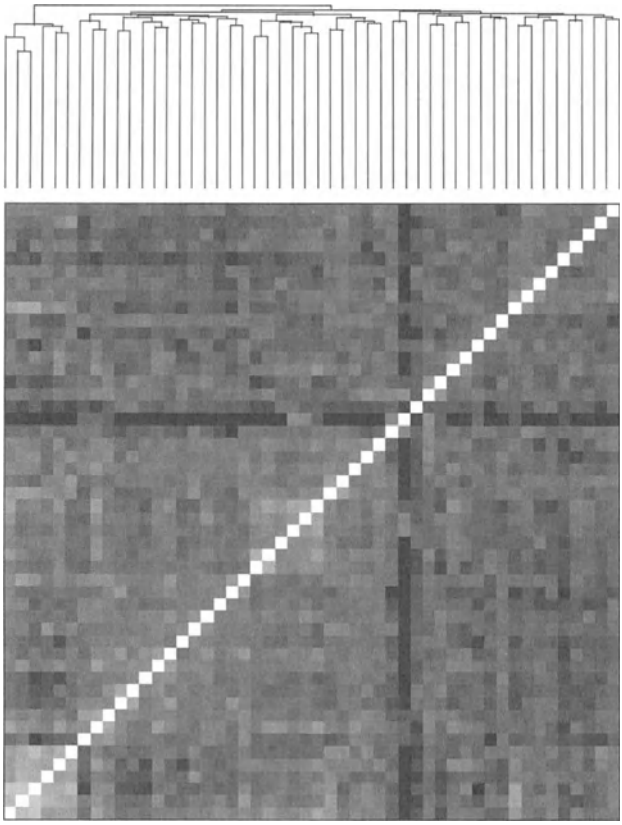

Fig. 5.9 Data image of the interpoint distance matrix for the data in Figure 5.7 using the binary distance on the vectors, which measures the number of times the same function occurs in the same place in the vector. The log of the distance is plotted as the intensity in this image.

them into "sessions" of length 100, as is done in Schonlau et al. [1999]. Then, for each user, we have 50 observations of dimension 100. We sort these observations within each user, using Equation (5.25) as the distance metric. Looking at the individual data images gives some reason for hope since they show that, for the most part, the within-user distances are small, indicating that under this distance the user sessions look fairly homogeneous. Nevertheless, as can be seen in Figure 5.12, there is some variability in the users' activity.

The data image for this ordering is shown in Figure 5.13. This is essentially the picture of Figure 5.6 with a smaller data set, which makes it a little easier to see the differences among users.

We have seen that there is some hope for a solution to the problem of detecting masqueraders, since the users do appear to be quite different from one another in their selection of commands and command sequences. However, the problem

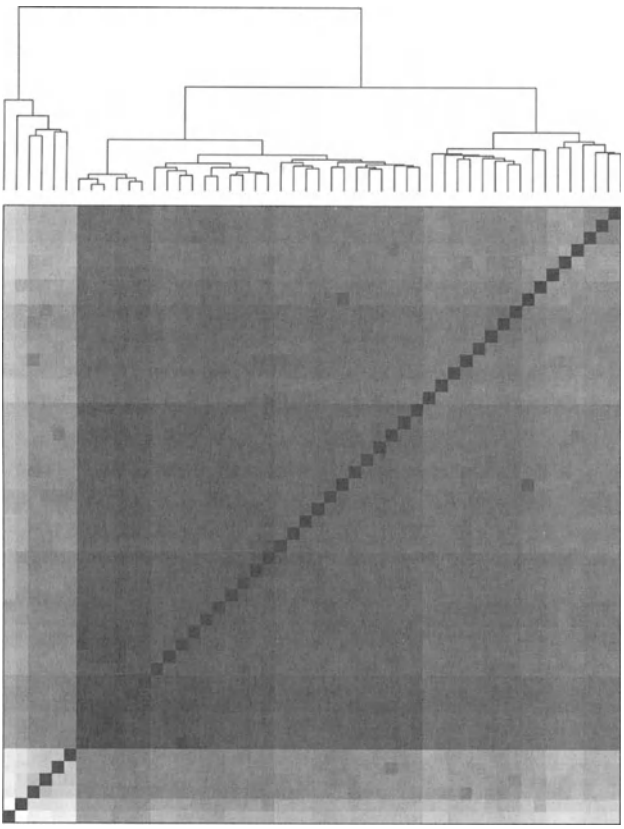

Fig. 5.10 Data image of the interpoint distance matrix for the data in Figure 5.7 using Equation (5.25) for the distance metric.

is still difficult due to the within-user variability and the difficulty of properly handling the temporal nature of the data.

Schonlau et al. [1999] report on a set of six methods for detecting masqueraders. I will describe these briefly, followed by a discussion of the results they report for the algorithms tested. The interested reader is urged to consult the paper for more details.

The authors devised a clever algorithm based on the following observation. There are many commands that are used by only one or a small number of users or are generally much rarer than other commands. These commands should be most useful for discriminating among users. The downside is that the commands must appear fairly often within any individual user's command stream or they become useless for detecting masqueraders in relatively short command sequences. They

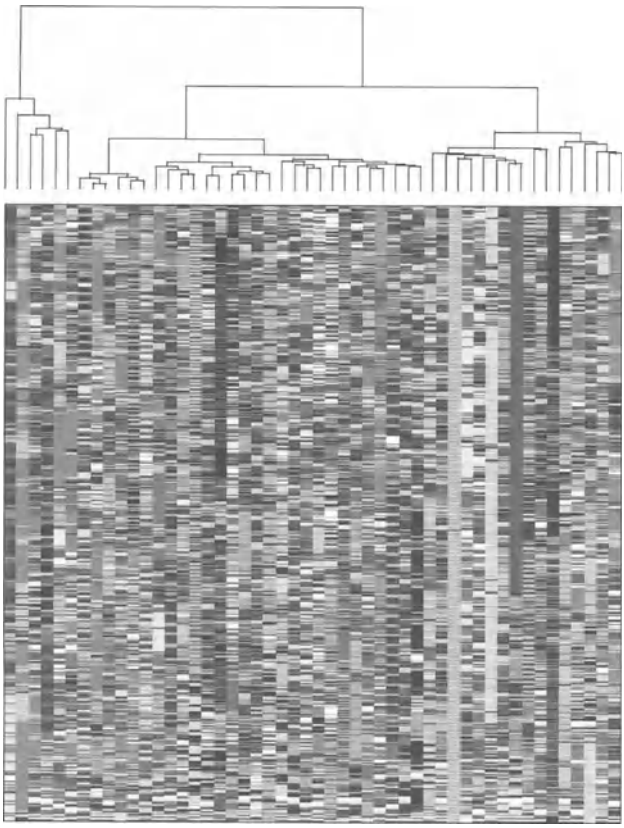

Fig. 5.11 Data image of the command sequences of Figure 5.7 using the user ordering produced by Equation (5.25) corresponding to Figure 5.10.

define a statistic

$$x_u = \frac{1}{n_u} \sum_{k=1}^{K} W_{uk}(1 - U_k/U)n_{uk}, \tag{5.26}$$

where U is the number of users, U_k is the number of users who have used command k in the training data, n_{uk} is the number of times command k appears in the block for user u, n_u is the number of commands in the block, K is the number of distinct commands, and

$$W_{uk} = \begin{cases} -N_{uk}/(N_u \sum_t N_{tk}) & \text{if command } k \text{ is in user } u\text{'s training data} \\ 1 & \text{otherwise.} \end{cases}$$

This scores the users based on whether they have used the commands in the block before, scoring rare commands more heavily than common ones. It ignores

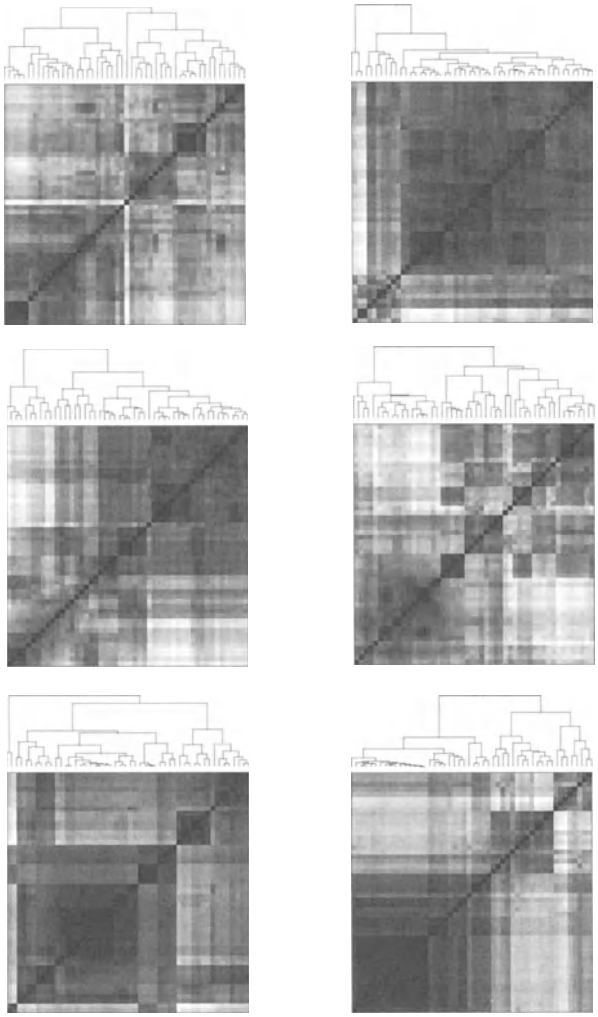

Fig. 5.12 Within-user distances for six users (users 1, 5, 19, 22, 35 and 43).

command ordering, being concerned only with a tally of the commands in the block. See Theus and Schonlau [1998] for some examples of using rare events to profile users.

A threshold (the same for all users) is calculated via cross validation. Then, for any new block, the statistic x_u is calculated and if it is above the threshold, the block is considered to be the result of a masquerader.

Another issue addressed in Schonlau et al. [1999] is updating. Although it is reasonable for a small data set to fix the training data, and hence the algorithm, and then test on the test set, this is not a realistic scenario for a deployed system. One wishes to use all the data available, and so a method for updating the algorithm is needed. The authors set a second threshold, and any blocks in the test set that fall

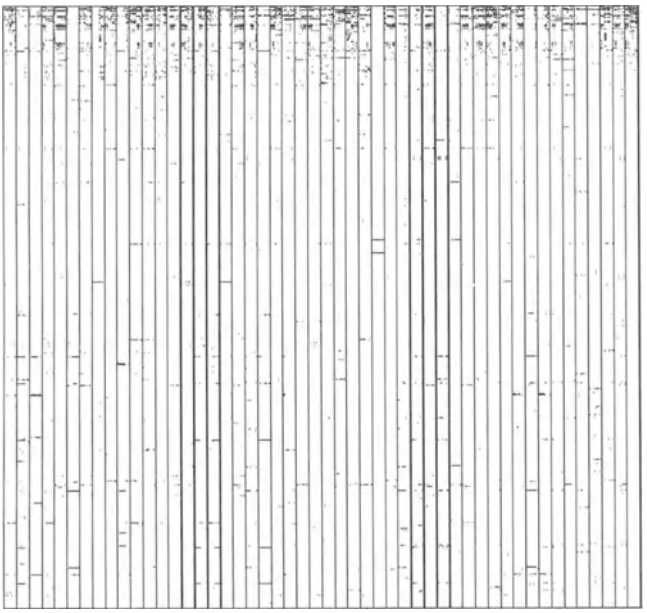

Fig. 5.13 Users ordered by Equation (5.25); compare with Figure 5.6.

below this threshold are added to the training data. This is done sequentially as the blocks are presented for scoring.

The next two methods considered are based on Markov models. These compute the transition probabilities. The first method, one-step Bayes, based on the work in DuMouchel [1999], computes the transition probabilities of executing command c_j given that the last command was c_i. This then tests the hypothesis that the matrix constructed on the test block came from the same process that produced the matrix computed on the training data.

Let C_t be the command observed at time t. Let p_{ujk} be the historical (estimated from the training data) transition probability from c_{t-1} to c_t. In other words, p_{ujk} is the probability that command t will be k given that the previous command was j. We need one further piece of machinery: the Dirichlet distribution. The vector of random variables $Q = (Q_1, \ldots, Q_K)$ has a Dirichlet distribution, with parameters $(\alpha_1, \ldots, \alpha_{K+1})$, if their joint probability density function is

$$f(Q) = \frac{\Gamma(\alpha_1 + \cdots + \alpha_{K+1})}{\Gamma(\alpha_1) \cdots \Gamma(\alpha_{K+1})} \times Q_1^{\alpha_1 - 1} \cdots Q_K^{\alpha_K - 1} (1 - Q_1 - \cdots - Q_K)^{\alpha_{K+1} - 1}. \quad (5.27)$$

The one-step method performs the following hypothesis test:

$$H_0 : P(C_t = k|C_{t-1} = j) = p_{ujk}, \tag{5.28}$$

$$H_1 : P(C_t = k|C_{t-1} = j) = Q_k. \tag{5.29}$$

The basic idea of the model is that one fits the p_{ujk} and the parameters of the Dirichlet distribution, α_k, using the training data. Details of these estimates can be found in DuMouchel [1999]. The hypothesis test involves computing

$$BF = P[C_1, \ldots, C_t|H_1]/P[C_1, \ldots, C_t|H_0]. \tag{5.30}$$

For large values of $\log(BF)$ from Equation (5.30), the null hypothesis is rejected in favor of a masquerader. More details about hypothesis testing for command transition probabilities are given in DuMouchel and Schonlau [1998].

The hybrid multi-step Markov uses a higher-order Markov model, when the data support it and a simpler model otherwise. This is described in some detail in Schonlau et al. [1999] and is based on work by Ju and Vardi [1999]. The basic idea is to expand the single-step model to a multi-step model, where care is taken to ensure that the model does not become unstable when the block contains many new commands. This is the "hybrid" part of the model, where this case is handled by a simpler model (a contingency table of users vs. commands). Again, refer to the papers cited for the details.

The next algorithm stems from the intuition that if new data are appended to the training data for a user, the augmented data will compress (nearly) as well as the training data, whereas if the new data are from another user, they will not compress well. they Given training data C and a block of test data c, the augmented data $\{C, c\}$ are constructed by appending c to the end of C. Using the Unix utility *compress*, the statistic compress$(\{C, c\})$ − compress(C) is computed. Large values result in a rejection of the hypothesis that the new data c were from the user associated with the training data C.

The next algorithm (IPAM) is similar to the one-step Markov model, except that instead of a hypothesis test, the transition probabilities are used to predict the next command, based on the current command. The number of incorrect predictions is tallied, and if this is large the data are flagged as a masquerader.

Finally, a method similar to the Forrest approach (see Section 5.4) is implemented. The idea is to consider all strings of commands of length 10 within the training data for each user. A similarity measure is defined where two sequences are compared command-by-command, with a score computed that is based on the number of matches, adjacent matches, and so on. Each new ten-command sequence is given the maximum similarity score from all the training data for the user. The most recent 100 consecutive such scores are then averaged, producing the score for the block.

Figure 5.14 depicts a data image of these ten-command sequences for the 50 users. Each user has 500 nonoverlapping blocks displayed as a data image. Since the blocks are disjoint, there are clearly many more command sequences for each user (in fact there are over 4000 distinct ten-command sequences in these data). One thing to note from this is that some users are clearly more homogeneous than others. Also, there are several users who are clearly distinct from others.

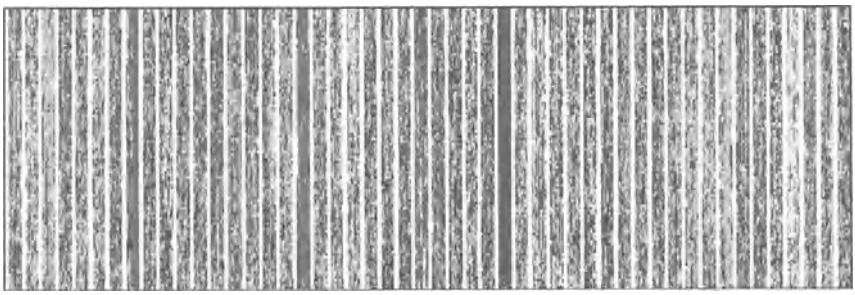

Fig. 5.14 Sequences of commands of length 10 for each of the 50 users are plotted as a data image. In this case, each user has 500 nonoverlapping blocks plotted. The white bands are separators between users.

The results of the six techniques are depicted in Table 5.3 and Figures 5.15 and 5.16. Both figures depict the results for the algorithms with updating. Similar figures are presented in Schonlau et al. [1999] for the algorithms without updating.

The test was blinded, so none of the researchers applying the algorithms knew the placement of the masqueraders or even whether there were masqueraders in a given user's data. The results reported in Table 5.3 are not too impressive, but as we have seen in our analysis, this is a particularly difficult task.

In Figure 5.15, each algorithm has been adjusted to produce a false alarm rate of 1%. Uniqueness appears to be the best algorithm by this measure, although IPAM is better in the region of lowest false alarm, which is where one wants the algorithms to operate. The probability of the intruder surviving (going undetected) is plotted against the number of blocks until a detection is made. Thus, we can see that, depending on the algorithm, between 5% and 30% of the attacks are detected. By the tenth block, somewhere between 20% and 40% have been detected. From this plot, it seems that IPAM and Uniqueness are the best algorithms.

The work of Schonlau et al. [1999] addresses only the commands executed. Ignored in this work are the arguments to the commands, in part due to the difficulty of dealing with these data. This leaves off several pieces of information that can be key to characterizing users. Some useful information that could be used in a follow-on study includes:

Table 5.3 The results (in percent) for the six algorithms reported in Schonlau et al. [1999]. The algorithms aimed at a false alarm rate of 1%.

Method	FA	PD
Uniqueness	1.4	39.4
Bayes 1-Step	6.7	69.3
Hybrid	3.2	49.3
Compression	5.0	34.2
IPAM	2.7	41.1
Sequence Match	3.7	36.8

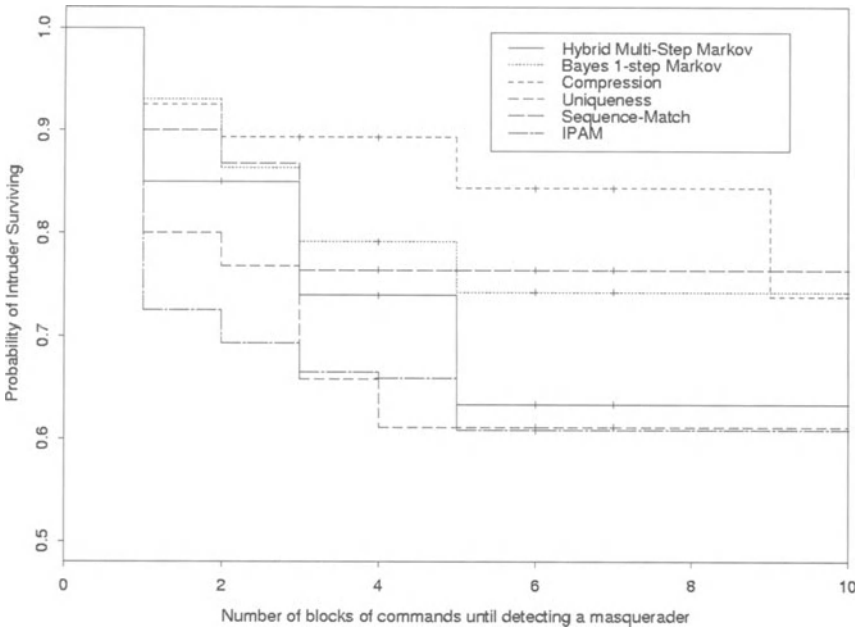

Fig. 5.15 Performance of the various algorithms tested in Schonlau et al. [1999] (courtesy of Matt Schonlau).

- **The command line flags used with the command.** For example, if I do a long listing ("ls -l"), I almost always combine this with a flag indicating the results should be sorted in time ("ls -lt"). This kind of information can be quite useful in characterizing users.
- **Whether the commands were executed from a script or otherwise.** A set of commands in a script may be executed by anyone with access to the script, and hence the order of execution of those commands is not really indicative of a particular user. However, some users like to put commonly executed sequences into a script for convenience.
- **The use of aliases.** To return to the "ls" example, I actually have an alias ("ll") for "ls -lt" and almost always actually type "ll" instead. In addition, I have a number of aliases that I set up thinking they were a good idea, but that I have stopped using (for various reasons). For example, to see the "new" files in a directory, one could type "ls -lt | head." I have an alias for this but never use it. I always type "ll | head." Why? Personal eccentricity. A masquerader, even if they looked in my .cshrc file (where aliases are defined), would not know which aliases I typically use and thus would have a difficult time matching my normal usage patterns.
- **Where the commands are executed.** More generally, which directories are visited and which files are touched. These can be both a tip-off that the user is a masquerader and an indication of what the user is doing and why.

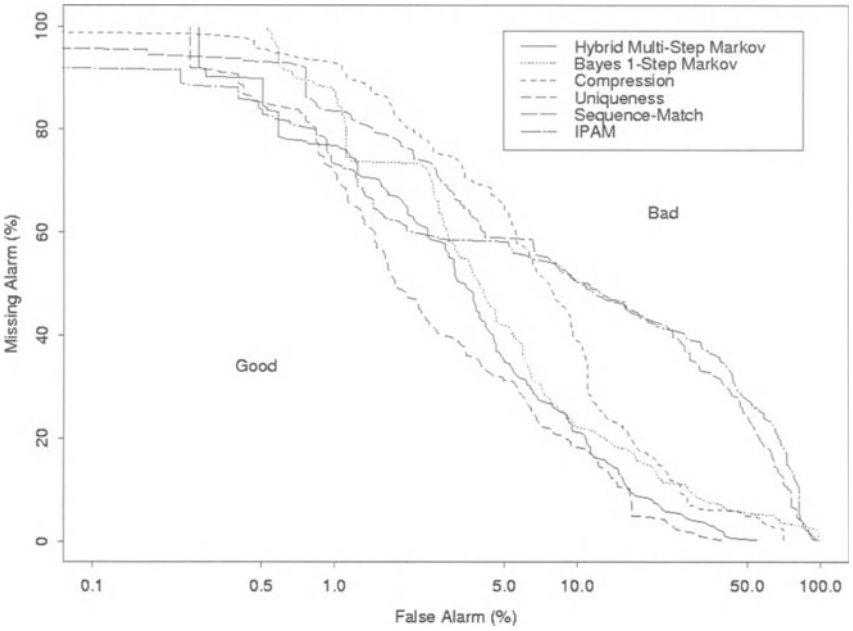

Fig. 5.16 ROC curves for the various algorithms tested in citeschonlau:1999 (courtesy of Matt Schonlau).

Further, even if the user is the authorized user he or she purports to be, this can be an indication of a "user gone bad," also known as "the insider threat."

- **The machines accessed.** As we saw earlier, one of the users simply used the machine as a terminal to (one or more) other machines. Other users may rarely go to other machines (through, for example, telnet, rlogin, FTP, ssh, etc.). However, if the user starts going to machines that have never been visited before, that is an indication that there might be a problem.

- **The shell used.** Users can be quite dogmatic about the shell they want to use, whether it be the bourne shell, the cshell, or one of the many others. Some users will automatically change to their favorite shell if they find themselves in another (as a result of accessing someone else's account).

- **Window usage.** Some people love the file manager windows and other "user-friendly" interfaces. Some people hate them.

- **How the user accessed the system.** If you never telnet to a machine, then anyone claiming to be you who telnets to the machine is suspicious.

This command-based detection can and should be combined with biometric information such as the keystroke timings. Other authentication techniques, such as key-cards or retina scans can be implemented.

The preceding discussion shows some of the promise of statistical methods for intrusion detection on host systems. Most of this work is quite recent, and there are

many areas for future research. The results to date are not that impressive given the desire for (near) perfect detection, but this appears to be in part due to the narrow focus of many of the projects. One of the areas for potential future work is to try to combine many disparate sources of information into a single set of algorithms. NIDES (Section 5.3) is a first attempt at this, and EMERALD (Section 4.6) is a more sophisticated attempt to do this.

5.6 MISCELLANEOUS UTILITIES

A few utilities for collecting data on a single host and monitoring for intrusions are described in this section. As always, this is not meant to be a complete list but rather a list of a few important utilities. More such utilities can be found in the resources listed in Appendix D.

5.6.1 strings

A useful program for investigating binary files (either programs or data) is "strings." This program prints out all the printable character sequences of at least 4 characters in the file. These may occur by accident - for example, if the data in the file just happens to have a value corresponding to the ascii values of the characters in the word "foobar." However, often there are error messages, prompts, version information, and so on in programs that are stored as character strings, and one can see these by running strings on the file.

For example, try running strings on ls:

strings /bin/ls | more

The use of "more" is to keep the listing from running off the screen. You will get a long listing of all the strings within the file. Scroll down the page, and you will come to the help information. This is what is printed out when you run

ls help.

When I ran strings, I got, near the bottom, the following strings:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
January
February
March
April
```

```
June
July
August
September
October
November
December
```

Note: there is no "May" because the strings program is looking for strings at least 4 characters long. Running strings on a program I wrote resulted in:

```
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
fscanf
calloc
fprintf
__deregister_frame_info
sscanf
fclose
stderr
exit
fopen
_IO_stdin_used
__libc_start_main
__register_frame_info
GLIBC_2.1
GLIBC_2.0
PTRhl
QVhP
@WVS
Usage:
sample
-i <infile> (stdin)
-o <outfile> (stdout)
-n <number_data_points>
-d <dimension>
-h this help
Unknown flag %s.  Try doubles -h for options
Could not open input file %s
Could not open output file %s
%d (%d):
%d (%d)
```

You can see the libraries called by the program, some of the functions used, and the help strings. There appear to be three strings that are accidental (for example, QVhP), a result of bytes that happen to have values within the range of printing characters. Running strings on a binary data file will display a large number of these accidental strings.

Some useful options to strings are:

- **-a** Scan the whole file (the default is to scan only the initialized and loaded sections of object files).
- **-m** *minlength* Print only strings of length at least *minlength*. The default is 4.
- **-t** Print the offset into the file before each string. A following o, x, or d defines the output radix as octal, hexadecimal or decimal.

5.6.2 ps and top

The ps command is used to report the processes running on a machine. For example, on my machine I just executed the command

ps aux

and obtained a long list of processes, a partial listing of which follows.

```
USER       PID %CPU %MEM TTY   STAT START TIME COMMAND
dmarche 10757  0.0  0.3 pts/0 S    08:23 0:00 -csh
dmarche 10758  0.0  0.3 pts/1 S    08:23 0:00 -csh
dmarche 10767  0.0  0.5 pts/2 S    08:23 0:00 -csh
dmarche 12376  0.0  0.4 pts/2 R    21:40 0:00 ps aux
```

This shows the process ID (PID), how much of the cpu and memory are used by the process, the status and start time of the process, and the command name (truncated if necessary for display). This could, in principle, be used to watch for attackers executing suspicious programs. However, it does not run continuously, so it is not a terribly good monitoring program. It is also relatively easy for a knowledgeable attacker to fool the ps program.

The "top" command performs a similar function. It runs continuously, updating the process list every second or so. It also sorts the processes by their CPU usage, which can be useful to determine who is hogging your machine. Top can be modified interactively (type the "h" key to see the options) and can be run in "secure mode" (run "top s"), which disables some of the interactive functions. The top man page suggests that running "top s" in a spare window is a "nifty thing" to do.

These utilities will show any processes that become active, provided someone (or some monitoring program) is watching. Unfortunately, it is possible to hide one's program so that it does not appear in a ps listing. On a related note, one of the first C programs I saw early in my career was a utility for changing the text that is displayed in a ps listing, so that instead of seeing something like:

dmarche 11232 0.0 0.3 5361 743 pts/0 S 09:32 0:00 crack /etc/passwd

ps might show something like

dmarche 11232 0.0 0.3 5361 743 pts/0 S 09:32 0:00 Your Ad Here!

Sufficiently sophisticated attackers (or those who have downloaded the right scripts) will simply hide their processes from ps, making it that much more difficult to detect their presence. Another approach is to run a "rootkit" that replaces the ps program with one that ignores the attacker's programs.

5.6.3 lsof revisited

We saw in Section 1.9.11 how to use the lsof utility to learn about the open Internet files and ports. We now turn to its use for host-based security. Lsof can be used to search for unlinked files, which are invisible to ls, using

lsof +L1

This lists the files with link counts less than 1, which may be files hidden by an attacker.

Another use for lsof is to collect data for user profiling. For example, using

lsof -u dmarche

will collect data such as

```
\symbol{44}\,ND   PID FD TYPE DEVICE   SIZE  NAME
Default   597 cwd  DIR    3,7   4096 /home/dmarche
Default   597 rtd  DIR    3,7   4096 /
Default   597 txt  REG    3,7 373176 /bin/bash
gnome-ses 610 cwd  DIR    3,7   4096 /home/dmarche
gnome-ses 610 rtd  DIR    3,7   4096 /
gnome-ses 610 txt  REG    3,7  46036 /usr/bin/gnome-
gnome-ses 610 mem  REG    3,7 344890 /lib/ld-2.1.2.so
```

where USER and NODE are not displayed in order to use a single line for the display. This provides information about which files the user opened as well as the application used to open them. This can be used both for profiling typical user activity and determining when a user is examining or using files that are not typically used by that user or are (supposed to be) off limits.

5.6.4 logcheck

The system logs are one of the most useful places for gathering data for use in intrusion detection. All problems and many diagnostics are reported to the syslog (in Linux these are in /var/log). On Sun Solaris systems, there is a utility called BSM, which provides extensive logging and auditing capabilities to the level of logging individual system calls if desired. These files get quite large, and there is a mechanism in all system loggers to roll the files over either at a specified time or a specified file size.

As can be imagined, it is a nontrivial matter to check the system logs for evidence of attacks. This begs for automation, and there are several such solutions available.

One of the easiest utilities for monitoring the system log is logcheck. This is a program that is started as a cron job (meaning that it is run at a prespecified time: every 15 minutes, once an hour, alternate Tuesdays, or whatever you want). It monitors the system log for changes, and then compares these new entries with a couple of lists of things you have told it about. These lists tell logcheck what to ignore, what to flag as definite attacks, and what to flag as suspicious. Anything that is not on the "ignore" list will be sent via email to wherever you specify. In particular, it can be sent to a corporate email account that is inaccessible from the machine if you are truly paranoid. (This is only useful if someone monitors this account regularly and so may not be a viable solution for some.)

With logcheck running every 15 minutes, I have found that I get something like 10–15 emails a day on my machine at work. One of the reasons I get so many is that I have several monitoring systems running, all reporting to the syslog, so I see a lot of things that I want to know about such as when others with accounts on my system log in. Most people are not this paranoid.

There are a number of system log monitoring utilities. One of the most popular is swatch, which can be found at

http://www.stanford.edu/~atkins/swatch/

I have found that logcheck meets my needs. It is relatively easy to configure and use. The fact that it emails reports to me as needed means that I can check up on my system when I am at home or traveling.

Logcheck can be found at:

http://www.psionic.com

5.6.5 portsentry

The portsentry utility is designed to watch for access attempts to a prespecified list of ports. It can watch either TCP or UDP ports (or two copies can be run, watching each protocol). It reports any access attempts to its list of ports to the system log.

In this manner, it is similar to the simple tcpdump-based monitor described in Section 4.5.4 on page 129. It is less flexible in what it detects, not having all the functionality of a full tcpdump filter, but it is quite powerful.

One of the interesting capabilities of portsentry is the ability to deny access to any host attempting to connect. This is done in two ways. First, an entry can be made in the /etc/hosts.deny file. This is a file that is checked when machines try to connect to your machine (assuming you have properly configured your machine). Any machine on the list is not allowed to connect. A second mechanism is to drop the route to the host. This basically places the host in your route table with a "block," forcing route lookups for the host to fail. Thus, no packets go back to the host, making it look as though your machine has disappeared.

I do not recommend this latter functionality for novices. It is too easy to accidentally drop the route to something important (like your router, for example).

These can be removed but not as easily as the entries in /etc/hosts.deny. Also, even if you manage to place your router in /etc/hosts.deny, no harm will be done (the router has no business logging on to your machine anyway). Luckily, the routing table is regenerated at boot time, so if you make too big a mess of it you can simply reboot. Thus, the dropped route stops the attack without doing any permanent damage. Dropping the route is probably a good thing to do for very important systems, or in "attack-rich" environments. Just be sure you don't allow an attacker to use it to effect a denial-of-service attack against your system.

Since portsentry sends its reports to the system log, you must either check your system log regularly or use a monitor program such as logcheck (Section 5.6.4). You should do this anyway.

Portsentry can be found at:

http://www.psionic.com

5.6.6 tripwire

One of the earliest file integrity checkers was tripwire, written by Gene Kim and Eugene Spafford of Purdue University. Originally, tripwire was designed to construct sophisticated checksums on a set of files to be protected. The checksums were stored on a removable medium. Periodically, one would compare newly computed checksums with those previously stored and report the files that have changed.

Tripwire also checks whether permissions have changed or whether the file modification times have changed. It records the deletion or addition of files within the directories it is protecting. Thus, it is useful for detection of attacks but also for assessing the consequences of the attack and the extent of damage. This makes tripwire, or a similar program, an essential tool for computer security.

Newer versions of tripwire allow the encryption of the checksum file, which allows it to remain on the protected disk. This makes periodic checks easier (a copy should always be kept on a removable medium to ensure that if the tripwire file itself is corrupted or removed one can still determine what else has been touched).

Tripwire first builds a database using a configuration file (/etc/tw.config) to indicate which files/directories are to be protected. The database is then stored away in a protected place. It can be updated if new files are added or if files are changed by the system administrator on purpose (for example, if a new user is added, the password file is changed). When an intrusion is suspected, or as a precaution against undetected intrusions, tripwire constructs a new database and compares it against the old. Any changes are reported to the operator.

Tripwire is "semi-free," in the sense that versions of it are available for free for noncommercial use. There is also a commercial product available. There are similar programs available, which include source code. One of these is aide, which stands for "Advanced Intrusion Detection Environment." One nice feature of aide is that it uses no shared libraries. (I believe that tripwire does not either but am not certain of this.) This is critical. As we will see in Section 7.6, one can modify the behavior of a program by modifying the libraries it loads. Thus, security tools should be very wary of the use of shared libraries if possible.

In any case, a file integrity checker such as tripwire or aide is an essential tool for computer security. As the manual for aide puts it, paranoia is your friend.

Aide can be found at:

http://www.cs.tut.fi/~rammer/aide.html

Tripwire can be found at

http:www.tripwire.org

or

http://www.sourceforge.net/projects/tripwire/

One might think that a file integrity checker is only useful after an attack or if one is paranoid enough to run the thing every once in a while as a check. It is possible, however, to run tripwire in a cron job, which executes it, for example, once a day and emails the results or puts them in the system log. In this manner, one has a daily check of the integrity of the system files. If one does this, it is essential that a copy of the tripwire database be kept on removable media and that a check still be made against this database periodically by the system administrator.

5.6.7 ipchains

In Linux, no discussion of security tools would be complete without the inclusion of ipchains. Although I make no claims of completeness, I will say a few words about this useful program.

Ipchains is an implementation of a firewall. It can provide a firewall in the usual sense, providing protection for a network, or it can be used as a host-based firewall, protecting your single host from attack. It is this latter use that I will discuss here.

A packet filtering firewall is basically a set of rules that tells the kernel what packets are to be let through to the IP stack. Any packets that pass the rule set are sent up the stack (or sent out on the network - the rules work both ways), and those that do not pass either generate an error or are simply ignored, depending on the rule. There is also a facility for logging that allows one to collect information on the packets that failed (or the packets that were let through, if one wishes).

There are two philosophies in packet filtering. The first, and the one that I recommend, is to deny everything except those few services that one wishes specifically to allow. The other, is to allow everything except those services that one considers a danger and specifically denies. With ipchains one can choose either approach, while with some commercial firewalls, only the first is allowed.

Setting up a personal firewall is far too complicated for this short section. Instead, I will illustrate what is involved with a small example. Some of the most commonly used arguments to ipchains follow.

- **-A|I [num]** Append the rule to the end or insert at rule "num."
- **-i interface** Set the interface to which the rule applies, for example eth0.

- **-p protocol** The protocol to which the rule applies, for example TCP.
- **-y** The SYN flag is set and the ACK and FIN flags are cleared. A "!" in front of this option inverts it: the SYN flag is not set, and the ACK flag is set.
- **-f** The rule applies to fragmented packets.
- **-s address:port** The source IP address and (if necessary) port or port range.
- **-d address:port** The destination IP address and (if necessary) port or port range.
- **-j policy** What to do if the rule matches a packet (ACCEPT, DENY, REJECT).
- **-l** Log any matches to the rule.
- **-L** List all the rules.
- **-F** Flush all the rules.
- **-P chain target** Set the policy (see the following).

An example will give some idea of how this works. Please do not try this without some further research. The man pages are a good place to start. An excellent book is Ziegler [1999]. Also, never execute these from the command line. Always put them in a script, and execute the whole script. One reason for this is that the first few will lock out all network accesses (including X Windows), and you may find that you cannot execute the others (or do anything else on your machine except reboot).

Let us look at a firewall that would let in only secure shell, ssh, and nothing else. This might be desirable on a network sensor, for example, but is probably too strict for most desktop workstations. It will, however, illustrate some of the ideas.

First, we set the policy:

```
ipchains -F
ipchains -P input DENY
ipchains -P output REJECT
ipchains -P forward REJECT
```

The first line flushes all current rules, leaving us with a clean slate. The second says that any packet sent to our machine will be ignored. The third and fourth say that we will not output or forward any packets, and we will generate an ICMP error message. Now we have blocked everything, so we need to carefully allow the things we want to allow in.

An important point here is that the first line removes all the current rules but does not change the policy. Thus, if you have the preceding policy in place, after

removing all the rules with the "flush" command, you are now in a position of denying everything. The distinction between rules and policy is important to remember.

```
ipchains -A input -i eth0 -p tcp -s any/0 1024:65535 -d 10.10.12.32 22 -j ACCEPT -l
```

This says that we will let any tcp packet from any IP address with a source port in the unprivileged ports 1024–65535 and destination port 22 (ssh) into our machine (IP address 10.10.12.32). The initial connection is logged, which is probably a good idea if you are at all paranoid. The preceding rule only lets packets in; we must let some out in order to have a connection.

```
ipchains -A output -i eth0 -p tcp ! -y -s 10.10.12.32 22 -d any/0 1024:65535 -j
ACCEPT
```

Note the "! -y," which allows everything but the SYN packet out. This is because these are precisely the packets that are needed for a connection initiated from the outside.

After the connection is initiated, ssh forks off a copy using privileged ports, starting at 1023 and going down. To allow up to five connections, we use

```
ipchains -A input -i eth0 -p tcp -s any/0 1019:1023 -d 10.10.12.32 22 -j ACCEPT
ipchains -A output -i eth0 -p tcp ! -y -s 10.10.12.32 22 -d any/0 1019:1023 -j
ACCEPT
```

To allow a different number of connections, change the port number range. If you want to also be able to secure shell out, you will need similar rules to allow these connections.

The "any/0" refers to "anywhere." If we want to allow only connections from our class B network 10.10, we could replace this with "10.10/16". This says that the first 16 bits of the IP address must match 10.10, and the rest can be anything.

Obviously, for a workstation there are many more rules to put in place, which is one reason for putting them all in a script. There are a number of utilities that help you write firewall rules, and the book by Ziegler [1999] is highly recommended.

5.7 FURTHER READING

There are several books and articles that provide insight into attackers' methods and motivation. Some famous ones include Stoll [1990], Cheswick [1992], and Freedman and Mann [1997].

Sekar et al. [1999a] suggest some methods for detecting race conditions. They provide several suggestions for process monitoring and describe a basic system for host monitoring.

Early work on IDES, the predecessor to NIDES, is discussed in Lunt [1989] and Lunt et al. [1990].

We have not really discussed visualization of host-based security in this chapter. Vert et al. [1998] suggest using "spicules," which are rays of decreasing thickness,

to represent various quantities such as CPU usage. Also, Jeffrey [1999] discusses several methods of program monitoring and visualization that would be relevant to host-monitoring.

An alternative approach to system monitoring is described in Elbaum and Munson [1999]. They model program execution as a stochastic system, where the probabilities of transitioning from one state to another (for example, calling function F given you have just called function E) are estimated from "normal behavior" and then used to detect abnormal activity.

Ghosh et al. [1999] propose using neural networks to model program behavior, with a goal of anomaly detection. Ko et al. [1994] and Ko et al. [1997] discuss the detection of vulnerability exploits in executing programs.

Ilgun [1993] describes a method of audit trail analysis based on state transitions to detect misuse. This allows the modeling of specific attacks as a series of transitions of state; for instance, whether a file is open or a link has been created. An inference engine then recognizes the type of attack that is under way. Mounji et al. [1995] discuss the analysis of audit trails from multiple machines or processes. In Mounji and Charlier [1997], they discuss integrating audit trail analysis with configuration analysis. Endler [1998] compares neural networks and a likelihood-based approach on Solaris Basic Security Module (BSM) data.

Bace [2000] also discusses state transition approaches, including work on colored petri nets. Chapter 4 of her book covers a large number of different approaches, including a number that I have not covered. It is a very good place to look for other approaches.

One of the problems security analysts have is the removal of sensitive information from the logs prior to disseminating the log, either for the purpose of reporting on the attack or providing researchers with data to analyze. Fisch et al. [1994] provide a brief discussion of these issues.

For a discussion of the legal and ethical issues related to intrusion detection, and what to do if you are attacked, several books have extensive discussions of these issues. Some in particular are Neumann [1995], Denning and Denning [1998], Denning [1999], Bace [2000], and Proctor [2001]. Another good book is Andrews and Peterson [1990], which provides insight into evidence gathering and analysis from the perspective of criminal investigations. A similar book is Casey [2000]. This latter focuses on the use of computers in crime and the issues of collecting and protecting digital evidence. There are a number of anecdotes of cases that make interesting reading as well as useful information. A readable overview for the layman is found in Icove [1997]. Brackney [1998] discusses the problem of intrusion response from the perspective of the U.S. Department of Defense.

The problem of user profiling continues to be of interest. Helman and Liepins [1993] propose a statistical method whereby pairs of the form *<user,action>* are modeled with probability of occurrence as a measure of misuse likelihood. Lane and Brodley [1997] discuss the issues of matching sequences of events for the purpose of user profiling or anomaly detection and show that exact matches are not appropriate for these purposes. They discuss a number of inexact matching algorithms. Büschkes et al. [1998] discuss anomaly detection in mobile networks, where user profiles are used to define "normal" activity for the user. A method for

profiling users of relational database systems is described in Chung et al. [1999]. This is used to detect misuses of the database, which can be indications of attacks against the database.

One aspect of user profiling that is a little different from what we have discussed here is the profiling of people by their Web activity. Martín-Bautista and Vila [2000] describe a technique to profile users by the documents they have accessed on the Web. The purpose would be to better serve, or market to, the users. An approach to clustering Web sessions is described in Nasraoui et al. [1999]. Another example of user profiling, concerned with providing personalized multimedia news and information to the user, is described in Tan and Teo [1998].

A good overview of the basic ideas of the computer immunology approach is found in Forrest et al. [1997]. This and related papers are available at

www.cs.unm.edu/~immsec/papers.html.

A technique that is similar to the immunological approaches described in Section 5.4 is discussed in Balasubramaniyan and Garcia-Fernandez [1998]. In this technique, autonomous agents are used as a hierarchy of intrusion detectors, which can be confined to a single host or migrate from host to host, providing protection for a whole network.

Miller et al. [2000] discuss a system that uses n-grams for document retrieval. The document is characterized by a vector of the n-grams that appear in it, and one retrieves those documents that are close to a given search string or example document.

The basic idea of computer immunology has also been used to detect attacks on Common Object Request Broker Architecture (CORBA) systems. This is described in Stillerman et al. [1999].

There are also several researchers applying machine-learning techniques to intrusion detection. For example, Lunt et al. [1989] and Bauer et al. [1989] discuss rule-based approaches, as do Snapp et al. [1991] and Lindqvist and Porras [1999]. Maloof and Michalski [1995] describe an inductive learning method. Shieh and Gligor [1991] discuss a method of detecting patterns of intrusions via finite-state machines. A data mining approach is described in Lee et al. [1998] and Lee et al. [1999b]. Rules are constructed using RIPPER to characterize various kinds of attacks, and association rules (such as "user1 reads email in the morning") are also utilized. Related work is described in Lee et al. [1999a]. A data mining approach to fraud detection is discussed in Stolfo et al. [1999]. Finally, Mé [1998] proposes a genetic algorithm to analyze audit trails and detect intrusions.

6

Computer Viruses and Worms

6.1 INTRODUCTION

Computer viruses are programs that copy themselves onto other programs. When the host program is run, the virus also runs, and as a consequence of its execution it makes further copies of itself. Most viruses also have other effects, such as erasing or damaging files, displaying rude words or pictures, or even damaging the computer or monitor itself.

The first computer virus, as reported by Cohen [1987], was created on November 3, 1983. It was an experimental program designed to demonstrate the possibility of virus programs, and it was released in a tightly controlled environment. It was extremely successful. This was predated by self-replicating programs written in the early 1970s at Xerox and the popular "core wars" games, where programmers wrote self replicating programs to compete for the resources (memory) of a computer. The first virus detected "in the wild" (as opposed to in the laboratory) was reported around 1985, according to McAfee and Haynes [1989] (Highland [1990a] places the first "wild" virus detection on October 22, 1987), so it didn't take long from the creation of the first virus to unauthorized infection of machines. A few early viruses are described in Highland [1990b], with many more described in McAfee and Haynes [1989]. A report on virus prevalence can be found at

www.trusecure.com/html/tspub/pdf/vps20001.pdf

Viruses most commonly infect personal computers. This is in part due to their general lack of any permission enforcement, meaning that any program can do pretty much anything on the machine in question. Another reason for this focus, particularly on Microsoft MS-DOS and Microsoft Windows machines, is that they

are so prevalent, and thus a virus can spread easily to many machines. This has been termed "death by monoculture," meaning that homogeneous environments are particularly susceptible to infection. This is true in biology as well, which is one reason why genetic diversity is usually a sign of health for a biological species or ecosystem.

In this chapter, we will first consider how viruses work. This tutorial is not aimed at providing instructions on how to write viruses but rather at providing a basic understanding of how they work. Following this is a brief discussion of how virus detection software works and some issues relevant to the statistician. Next, we will consider some work to model the spread of computer viruses, using random graph models. We will then look at some extensions of the immunological ideas discussed in Section 5.4. The final section on computer viruses will look at some work on producing computer virus phylogenies.

We then turn to computer worms. The distinction between worms and viruses is subtle, and some would say irrelevant, but worms can be quite different from viruses. In particular, worms can be stand-alone programs, whereas viruses are usually attached to other programs. We will look at a few famous worms and discuss some issues related to their detection.

The definition used in this book for a worm is that it is a program that spawns running copies of itself. An alternate definition is that a virus requires the action of a human to replicate, while a worm does not. The distinction is not really that important, in my view. While I will categorize individual programs as worms or viruses (or both), I understand that some will disagree with these categories.

Actually, if you think about it, the two definitions are not that far apart. A virus that infects a program requires the program to be executed (hence requiring user intervention to propagate). The differences will be more pronounced in some cases than in others, and I leave it to the reader to decide whether it's important enough to be pedantic over.

6.2 VIRUS REPLICATION

In order to understand how viruses replicate, we need to consider some details about computer programs. A computer program is nothing more than a section of memory in the computer containing binary words that are codes for performing certain actions. As a result, one can change a program simply by writing new values into the part of memory in which the program resides. This is how viruses propagate - by copying themselves into other programs. Usually, computer viruses operate on programs stored to disk rather than in memory, but the principle is the same.

Most people who have used computers for any length of time have come across the concept of a "software patch," which is software that vendors give out to fix problems in software the user has already purchased (see Ali [1991]). Considering how patches work will help in understanding how viruses replicate.

In its simplest form, consider the program illustrated in Figure 6.1. The patch program is given the instructions to allow it to find the beginning of the code to be patched. It places a "jump" instruction here, so that when the execution reaches

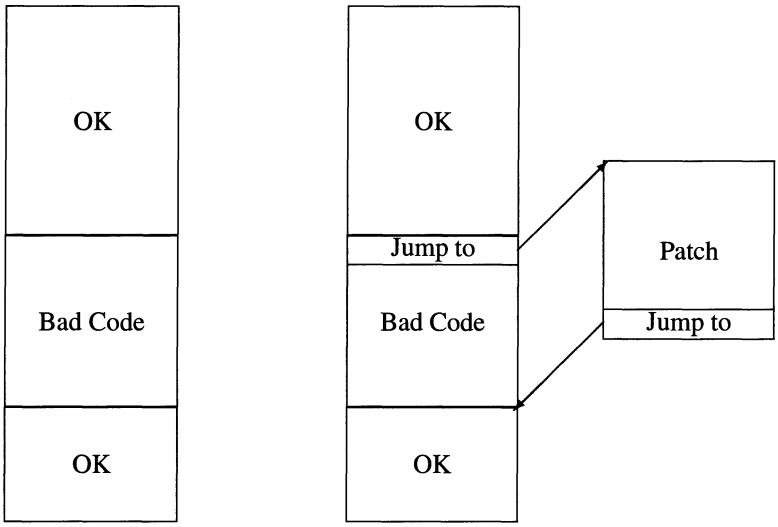

Fig. 6.1 Patching code. A "jump" instruction is placed at the beginning of the "bad code" causing execution to move to the patch code. At the end of the patch is another "jump" instruction to return to the beginning of the remaining "good" code.

this position in the code it will jump to the patch code. The patch program then places the appropriate return code at the end of the patch so that execution returns to the beginning of the remaining code.

This is a simplification. A real software patch may need to save and restore various registers to ensure that the code functions properly before and after the patch code. Also, it is possible for other parts of the program to have commands that cause the execution to jump to somewhere inside the "bad code." All such cases need to be handled properly. Further, all the patches that have been applied to a particular program need to be known by the patching software since the patches and the order in which they are applied can impact the manner in which new patches are to be applied. This is beyond the scope of this discussion and is irrelevant for virus propagation.

Obviously, the patch code resides somewhere in memory, and we could illustrate this by placing it on the bottom of the code depicted in the figure. Alternatively, if the patch is no larger than the "bad code" block, we could place the patch in the memory that contained the bad code, overwriting what is there with the patch.

One can think about virus replication as if the virus is "patching" the code with its own code. A description of some methods of virus replication can be found in Davis [1988]. Some sample code is provided in that paper to show how some of the basic ideas can be implemented.

One of the things that a virus needs to do is hide its presence to avoid detection. For example, every time a file is changed, its date field is changed, so a virus will generally change the date field back to its original value. Some viruses go to pains to ensure that the size of the file does not change.

If the infected program does not operate normally, the probability of detection increases, so most viruses try to ensure that they do not change the program execution in any noticeable way. The simplest way to do this is to place the virus at the beginning of the execution code. When the program is run, the virus executes first, propagating itself, or executing whatever other actions it is designed to perform. Then, the original program is executed as if the virus wasn't there. Some viruses will even erase their presence from the original program once they have successfully propagated.

6.3 VIRUS SCANNERS

There are two changes that are unavoidable: the actual values in the file must change, and the program execution must change to allow the virus to propagate (and potentially cause damage). It is these changes that allow virus detection software to detect viruses.

There are basically four ways to detect viruses. A brief discussion of these ideas can be found in Kumar and Spafford [1992], which also includes an implementation of a virus scanner.

One approach is to keep checksums for every file (or every file under your protection) and provide a notification to the system administrator when the checksums have changed for any file. This must be done with some care. Obviously, the stored checksums cannot be modifiable since otherwise the virus could simply change the stored value to match the file's new checksum. In fact, the stored checksums cannot even be readable or computable by the virus because if they were the virus could simply add data to ensure that the checksum matched the originally computed value.

One way to ensure that the stored checksums are not altered is to store them on removable or read-only media. This is the most secure. An alternative is to encrypt the checksums and store the encrypted values. Neither of these approaches directly addresses the last point, which is that if the virus can compute the checksum then it can ensure that the final checksum matches the original. This can be arranged by making the checksum calculation depend on an external event, such as a password. For example, the simplest approach would be to have the word-size used in the checksum be user-settable and use a different one for each file or even for regions within a file. The resulting checksum algorithm could then be encrypted and stored along with the encrypted checksum.

Of course, the ultimate version of this approach would be to keep a complete copy of the file either on a removable media or encrypted. Then the file can be compared bit-for-bit to ensure that no changes have occurred. This cannot be defeated, (assuming both the operating system and the comparison program are also kept on read-only media), but what it makes up for in security it loses in convenience, time, and storage requirements. This is not an unusual tradeoff in computer security.

There are versions of operating systems that are run entirely from a CD ROM. Since the CD ROM is read-only, these are safe from virus infection, at least for the operating system files on the CD. Of course, CD ROMs are not as fast as hard

disks, nor can they contain as much data, so these operating systems are necessarily smaller than those that are stored on the hard disk.

A second approach to virus detection is to compile a list of "signatures" of known viruses and search for these signatures within the files. This is the approach taken by virus detection software, and its widespread use attests to its success. A particular virus will place certain code in given (usually fixed) places within the file, and searching for these distinct patterns allows the detector to find and remove the virus. Of course, the downside to this is that as the number of viruses (and hence signatures) increases, the probability of false detections increases as well.

A technique related to the signature approach will be discussed in Section 6.5. The idea is to construct a set of "nonself" signatures, which are designed to detect sections of code that are not "normal."

Several potential defenses against the signature approach are possible for the virus writer. Most viruses place their code at the beginning or end of the file. Instead, the virus can be designed to place code at random places within the program. This requires the anti-virus software to scan the entire file to detect infection. On the other hand, it makes it much more difficult to hide the effect of the virus if the program is executed. This may seem irrelevant since once the program executes the virus has propagated, but it can inform the system administrator that the system has become infected and steps can be taken to isolate the computer from others, stopping further infections. Also, as will be seen later, one method of virus detection relies on running the program in a protected environment, so unusual behavior will produce a detection without allowing the propagation of the virus.

The virus can compress or encrypt the code prior to placing it into the program, making the actual byte values inserted change with each infection. This means that the main body of the virus cannot be detected by a scanner; however, the encryption/decryption part of the code must remain, and this can form the basis for a detector.

Finally, viruses can be written to change their own code - so-called "polymorphic" viruses. These programs mutate, causing their copies to be different from the "parent" virus, making their detection via scanning for signatures much more difficult or impossible. The "polymorphic" engine must remain to some extent, however, and this can form the basis for a detector.

The third approach to virus detection is to categorize "bad behavior" and scan programs for evidence that they are engaging in such behavior. For example, very few programs should be allowed to write on the boot sector of a disk. Most should not be allowed to format hard-drives, delete files outside of their working directory, write to system memory, and so forth. An extreme version of this is what is called "generic decryption" technology (Nachenberg [1997]). The idea is to build an emulation of the computer, run the program within the emulation, and determine whether it performs any of the actions that are proscribed. This is an extremely powerful idea, but it has some drawbacks. First, it is only as good as the emulation. It might be possible to write a virus to detect an imperfect emulation and simply not activate (copy itself or run) if it is being run by an emulation. Also, one does not want to run in emulation all the time, so, for example, a virus that is designed

to activate only on the tenth execution of a program, or on December 13, 2003, may go undetected.

The fourth approach is to consider statistical measures for "normality" to determine when programs are not acting as they should. This is similar to the approach taken in host-based intrusion detection, discussed in Section 5.4.

The preceding discussion has focused on viruses that infect files. These are the traditional viruses that most people who owned personal computers in the 1980s and 1990s recognize. There is a new breed of viruses, called macro viruses. There was a time when it was said that a virus could not be transmitted via email. There were a number of hoaxes, the so-called "Good Times Virus" and others (Denning [1999], pp. 276–279). In these, an email message was sent warning against reading certain emails that would infect your machine with a virus, erase your hard drive, or cause other catastrophic events (there was even an urban legend about some people fearing that biological viruses could be sent via email). Of course, all of this was nonsense. You could not get a virus (computer or otherwise) by reading email.

This is no longer true. As the Melissa virus (actually a worm by our definitions; see Section 6.7.3.1) showed (Garber [1999]), simply opening an attachment to an email can cause unforeseen consequences. Word processors now come with very powerful macro-languages that allow extremely sophisticated operations to be performed merely by opening a document.

A related issue is the increased power and functionality of email readers. As vendors increase the functionality of these readers (for example, automatically displaying attachments or executing mobile code sent by email), the possibilities for virus infection will be increased. If past is prelude, the security issues are not likely to be adequately addressed in these programs until the next big event causes users to demand protection.

The effectiveness of viruses is a function of two factors: the number of machines that can be infected by the virus and the susceptibility of the machines to damage. Unix machines have not been as common a target of viruses as personal computers due to their file protection scheme and other security measures, which limit the damage that simple viruses can do. As personal computer operating systems start to adopt security measures, viruses will have to be more sophisticated in order to get around the security measures, making them harder to write, and thus, presumably, less common. Another reason few viruses were written for Unix platforms was the diversity of such platforms. A virus that would infect a Silicon Graphics workstation would be unable to propagate to a Sun workstation, for example. The very strength of the personal computer (relative interoperability across all PCs) is a fundamental reason for the success of virus attacks.

Cohen [1987] proved that a perfect virus detection scheme is impossible. The proof is as follows:

Let D be a perfect virus detector; that is, for a program P we have

$$D(P) = T \quad \text{if } P \text{ is a virus}$$
$$D(P) = F \quad \text{if } P \text{ is not a virus.}$$

Let the program V be defined as follows:

if $(D(V) = F)$ then infect another program
else do nothing

If $D(V) = F$, then V is not a virus, yet V then proceeds to infect another program, thus proving that it is in fact a virus. Thus, we must have $D(V) = T$, in which case V is a virus but it does nothing and hence is not a virus. This is a contradiction.

This seems at first glance to be somewhat silly, an impractical "non-existence" proof. However, it is precisely the strategy that I suggested to defeat the generic decryption technology described in Nachenberg [1997]. First, the virus checks to see whether it is running in an emulator - if so, it does nothing; otherwise, it infects. Given that the virus writer can buy the same anti virus software that you can, it will always be possible for a sufficiently clever virus writer to defeat any given detector.

For another discussion of virus detection and elimination, see Phillippo [1990].

6.4 VIRUS EPIDEMIOLOGY

Treating computer viruses as if they were biological ones and studying their behavior using the tools of epidemiology is an appealing idea. Kephart et al. (Kephart and White [1991], Kephart and White [1993], Kephart et al. [1993]) describe this approach, and we will describe their work in some detail in this section.

Consider the problem of describing the spread of a virus over the Internet. The idea is to consider the Internet as a number of hosts each of which can pass a virus (assuming they are infected) to a fixed set of hosts.

First, some terminology:

virus A computer virus is a program that copies itself (infects) to other programs. It may or may not perform other tasks.

worm A computer worm is a program that spawns running copies of itself.

infected A computer is infected if the virus exists on the computer (in a manner such that the virus can be run or passed to another computer).

susceptible A computer is susceptible to a virus if it could become infected with the virus, provided the virus is somehow introduced to the computer.

adequate contact Two computers have adequate contact if one would have transmitted a virus to the other had it been infected and had the other been susceptible.

birth rate The birth rate of a virus is the frequency with which adequate contact occurs.

cured A computer is cured of a virus if all copies of the virus are removed from the computer.

death rate The death rate of a virus is the frequency of cure.

epidemic An epidemic is the widespread occurrence of a disease.

epidemic threshold The epidemic threshold is the relationship between the birth rate and the death rate at which the virus becomes widespread.

extinction rate The extinction rate is defined to be the ratio of the death rate to the birth rate.

endemic A disease that can maintain an epidemic for a long time is called endemic. For example, common childhood diseases are endemic.

Let us consider a simple epidemiology model, the SIS Susceptible-Infected-Susceptible model, depicted in Figure 6.2, in which a computer is susceptible to infection until it becomes infected. Subsequently, the computer may be cured, at which point it is once again susceptible to infection. There is no concept of immunity in this simple model. All computers are either susceptible or infected.

A compartmental model (Walter and Contreras [1999]) is constructed by dividing the system into homogeneous entities (compartments) and identifying the flow between compartments. Thus, for the SIS model, we have two compartments, S and I. The flow from S to I corresponds to the changing of a computer from susceptible to infected. This happens by an infected computer interacting with a susceptible one and so happens at a rate proportional to the product of the number of infected machines and the number of susceptible machines. Machines that are infected become susceptible (cured) at a rate proportional to the number of infected machines. This is illustrated in Figure 6.2.

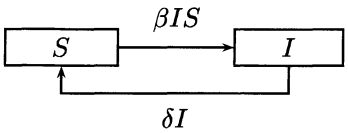

Fig. 6.2 The compartmental model for an SIS epidemic (see Walter and Contreras [1999]).

Setting $N = S + I$, this results in the differential equation

$$\frac{dI}{dt} = \beta I(N - I) - \delta I. \tag{6.1}$$

This has the solution (Walter and Contreras [1999])

$$I = \left(N - \frac{\delta}{\beta}\right) / \left(1 + Ce^{-(\beta N - \delta)t}\right), \tag{6.2}$$

where C is a constant depending on the number of initially infected computers.

Kephart and White [1991] consider the SIS model as well but take a slightly different perspective. The network is modeled as a random graph G with $\|G\| = N$

nodes. For each pair of nodes, a random, independent decision is made as to whether the nodes are connected. These connections are directional, to denote the fact that in some cases infection can only travel one way due to, for instance, the security policy on one of the nodes. Recall that graphs with directed edges are called digraphs.

There are $N(N-1)$ possible (directed) connections. Let P be the probability of connection, fixed for all pairs of nodes. Let $e(G)$ denote the number of edges in a graph (also called the size of the graph). Then, we have

$$E[e(G)] = PN(N-1),$$

where E denotes the expectation with respect to the probability model.

Let the infection rate from node j to node k be denoted $\beta_{jk} = \beta$. Similarly, let the death rate be denoted $\delta_{jk} = \delta$; that is, the rates are the same for all nodes. $I(t)$ will denote the number of infected nodes at time t and define the fraction of infected nodes $i(t) = I(t)/N$. For a given node, the expected number of edges from the node is

$$E[\text{number of edges from node}] = P(N-1) \equiv \bar{b}.$$

The fraction of neighbors to the node that are susceptible is $1 - i(t)$. The expected number of nodes that can be infected by this node is $\bar{b} * (1 - i(t))$. Thus, on average, we expect the total rate by which infected nodes infect new nodes to be

$$\beta I(t)\bar{b}(1 - i(t)) \equiv \beta' I(t)(1 - i(t))$$

Similarly, the rate at which nodes are cured is $\delta I(t)$. This results in the deterministic differential equation

$$\frac{di}{dt} = \beta\bar{b}i(1-i),$$

which is essentially Equation (6.2), and which the authors solve as

$$i(t) = \frac{i_0(i - \frac{\delta}{\beta'})}{i_0 + (i - \frac{\delta}{\beta'} - i_0)e^{-(\beta'-\delta)t}},$$

where i_0 is the initial fraction of infected nodes.

If we consider the ratio of cure rate to death rate, $\rho' = \delta/\beta'$, then

$$i(t) = \frac{i_0(i - \rho')}{i_0 + (i - \rho' - i_0)e^{-(\beta'-\delta)t}},$$

and this simple analysis leads to two cases of interest. If $\rho' > 1$, it is easy to see that this corresponds to exponential decay with $i(t) \to 0$. If $\rho' \leq 1$, then $i(t)$ asymptotes to $1 - \rho'$. This matches our intuition: if the death rate is greater than the birth rate, the virus eventually dies out; otherwise, it reaches an equilibrium.

Ignoring the stochastic nature of the problem leads to a very simplistic model. However, the results are not uninteresting. Basically, we have a crude bound for

the $i(t)$ depending on ρ'. Either $i(t)$ goes to 0, which means that the virus becomes extinct, or it is roughly bound by the value $1 - \rho'$.

In a stochastic model, it is easy to see that the virus must always become extinct since at any time there is a nonzero probability that every infected node will be cured. This is another reason to consider a more reasonable model.

Let $P(I,t)$ be the probability distribution function for I infected nodes at time t. Then, the probability of extinction at time t is $P(0,t)$. Let

$$\begin{aligned} I_- &= I - 1 \\ I_+ &= I + 1, \end{aligned}$$

and define $R_{a\to b}$ to be the rate at which transitions occur from state a to state b. By considering the nodes that become infected versus the nodes that become cured at time t, Kephart and White derive the equation

$$\frac{dP(I,t)}{dt} = -P(I,t)[R_{I\to I_+} + R_{I\to I_-}] + P(I_+,t)R_{I_+\to I} + P(I_-,t)R_{I_-\to I}. \tag{6.3}$$

$R_{a\to a+1}$ is calculated as

(#infected nodes)*(rate of infection)*(probability of susceptibility)*(#of neighbors):

$$\begin{aligned} R_{a\to a+1} &= a(1 - \frac{a}{N})\beta\bar{b} \\ &= a(1 - \frac{a}{N})\beta'. \end{aligned}$$

The probability of cure is (# infected)*(cure rate):

$$R_{a\to a-1} = a\delta.$$

Letting $i_- = I_-/N$, Equation (6.3) becomes (suppressing the dependence of I and i on t)

$$\frac{dP(I,t)}{dt} = -P(I,t)[I(1-i)\beta' + I\delta] + P(I_+,t)I_+\delta + P(I_-,t)I_-(1-i_-)\beta'. \tag{6.4}$$

For N nodes this is a tri-diagonal set of $N+1$ coupled linear differential equations and hence relatively easy to solve. If we define the matrix A with $a_j, j \in 0,...,N$ on the diagonal, $b_k, k \in 0,...,N-1$ above and $c_l, l \in 1,...,N$ below the diagonal, we have

$$\begin{aligned} a_j &= j(1 - j/N)\beta' + j\delta \\ b_k &= (k+1)\delta \\ c_l &= (l-1)(1-(l-1)/N)\beta'. \end{aligned}$$

These equations can now be written as

$$\mathbf{P}' = \mathbf{AP},$$

which can be solved (see Kreyszig [1999], pp. 162–163) as

$$\mathbf{P} = \alpha_0 \mathbf{x}^{(0)} e^{\lambda_0 t} + ... + \alpha_N \mathbf{x}^{(N)} e^{\lambda_N t}, \tag{6.5}$$

where the λ_i are the eigenvalues of $\mathbf{A}$ and the $\mathbf{x}^{(i)}$ are the corresponding eigenvectors.

For the simplest (non-trivial) case, where $N = 2$, we have

$$\mathbf{A} = \begin{pmatrix} 0 & \delta & 0 \\ 0 & -(\frac{1}{2}\beta' + \delta) & 2\delta \\ 0 & \frac{1}{2}\beta' & -2\delta \end{pmatrix},$$

and solving for the eigenvalues and eigenvectors of A produces

$$\lambda_0 = 0, \tag{6.6}$$

$$\mathbf{x}^{(0)} = (1,0,0)', \tag{6.7}$$

$$\lambda_1 = -\frac{1}{4}(\beta' + 6\delta + \alpha), \tag{6.8}$$

$$\mathbf{x}^{(1)} = (\frac{2\delta(\beta' - 2\delta + \alpha)}{\beta'(\beta' + 6\delta + \alpha)}, -\frac{\beta' - 2\delta + \alpha}{2\beta'}, 1)', \tag{6.9}$$

$$\lambda_2 = -\frac{1}{4}(\beta' + 6\delta - \alpha) \tag{6.10}$$

$$\mathbf{x}^{(2)} = \frac{2\delta(\beta' - 2\delta - \alpha)}{\beta'(\beta' + 6\delta - \alpha)}, -\frac{\beta' - 2\delta - \alpha}{2\beta'}, 1)', \tag{6.11}$$

where

$$\alpha = \sqrt{\beta'^2 + 12\beta'\delta + 4\delta^2}.$$

Using the fact that $\mathbf{P}(0) = (0,1,0)'$, we can use Equation (6.5) and Equations (6.6–6.11) to solve for $\alpha_0, \alpha_1, \alpha_2$:

$$\alpha_0 = 1,$$

$$\alpha_1 = -\frac{\beta'}{\alpha},$$

$$\alpha_2 = \frac{\beta'}{\alpha}.$$

The resulting solution of Equation (6.4) for $N = 2$ is then

$$P_0(t) = 1 - \frac{2\delta(\beta' - 2\delta + \alpha)}{\alpha(\beta' + 6\delta + \alpha)} e^{-\frac{1}{4}(\beta' + 6\delta + \alpha)t} + \frac{2\delta(\beta' - 2\delta - \alpha)}{\alpha(\beta' + 6\delta - \alpha)} e^{-\frac{1}{4}(\beta' + 6\delta - \alpha)t},$$

$$P_1(t) = \frac{\beta' - 2\delta + \alpha)}{2\alpha} e^{-\frac{1}{4}(\beta' + 6\delta + \alpha)t} - \frac{\beta' - 2\delta - \alpha)}{2\alpha} e^{-\frac{1}{4}(\beta' + 6\delta - \alpha)t},$$

$$P_2(t) = -\frac{\beta'}{\alpha} e^{-\frac{1}{4}(\beta' + 6\delta + \alpha)t} + \frac{\beta'}{\alpha} e^{-\frac{1}{4}(\beta' + 6\delta - \alpha)t}.$$

Figure 6.3 depicts the solution for $N = 2$.

Although this is a trivial example, one can still see interesting behavior, which is typical of the solutions for more realistic values of N. In Figure 6.3 we see that the probability of extinction starts at zero and quickly goes toward an asymptote of 1. Similarly, the probability of both systems becoming infected quickly increases to a maximum and then drops off toward zero.

Since the probability of extinction approaches one, the limiting distribution is an uninfected network. If we consider the conditional distribution of infection given that at least one of the nodes is infected, we have a limiting distribution that has the probability of both nodes being infected at slightly over 0.6 (when $\beta' = 1$ and $\delta = .2$):

$$\lim_{t\to\infty} P(\text{both infected}|\text{one infected}) = \frac{2\beta'}{\alpha + 2\delta + \beta'}. \tag{6.12}$$

The same kind of analysis can be performed in the general case. Analyzing Equations (6.4) and comparing the results to the deterministic model described earlier (with $N = 100$, $\bar{b} = 5$, $\beta' = 1$, and $\delta = .2$), Kephart and White [1991] make the following observations, which agree with our simple example. The density considered is the mixture of the densities $P_i(t)$ at a fixed t

- At time $t = 0$, there is only a single machine infected, so the density is a delta function at $I = 1$.
- As time progresses, the density is a mixture of a delta function at $I = 0$, corresponding to the extinction of the virus, and a "survival" component.
- At time $t = 1$, the survival component is exponentially distributed.
- At first, the survival component grows quite rapidly (exponentially), with a growing standard deviation. This continues until the population becomes saturated, reaching a balance between new infections and cures.
- The survival component at saturation is roughly Gaussian. They call this the "metastable" phase.
- The metastable phase is long-lived. However, the extinction component grows slowly until finally the virus goes extinct.

The preceding discussion shows that the probability of a virus going extinct eventually is 1. However, it is possible to analyze the metastable distribution. The conditional probability of I infections given that there is at least one infected machine does approach a well defined limit, which Kephart and White [1991] denoted $p_\infty(I)$. In our simple example, the metastable distribution is indicated in Equation (6.12).

This model is, of course, a simplification of the true problem. Several possible extensions to this model are discussed in Kephart and White [1991] and Kephart and White [1993]. Some of these, and a few suggestions of my own, are presented in the following list:

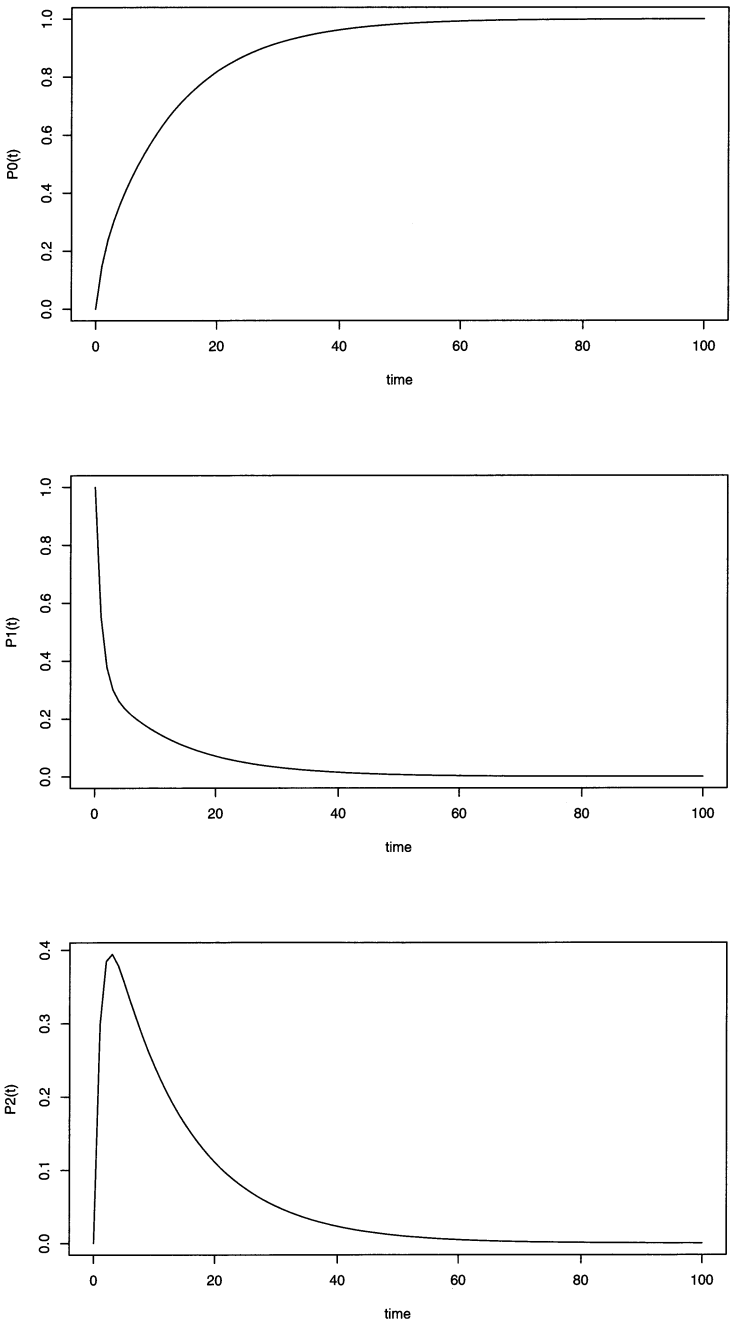

Fig. 6.3 Solution of Equations (6.4) for $N = 2$ with $\beta' = 1$ and $\delta = 0.2$.

- People tend to share files often with a small group but also occasionally with a much larger group. This could be modeled as a hierarchical model, where a machine has a small group under it that has a high probability of infection, then a larger group under that with a smaller probability of infection, and so on. This is a special case of the more general case where β_{jk} is not constant.

- Similarly, not all machines have an equal probability of cure.

- In some cases, a spatial model could be considered, where each node can infect others in its neighborhood but none outside the neighborhood. This could be a model for some kinds of security systems where machine interactions are restricted to those within a given security domain, with gateways between domains.

- Virus spreading in an organization is another interesting model. In this model, there is a collection of machines that can communicate with each other and a "boundary" through which all communications to the Internet must pass. In this model virus infections can be inserted only through the boundary, and thus even after extinction within the organization there is a probability of re-infection via machines outside the organization.

- Even after a virus is extinct "in the wild," it can still be re-introduced by attackers or by accident from archives. This can be modeled either by the organizational model described earlier, by having a small probability of "spontaneous infection" whereby a node becomes infected even though it is not susceptible from any connecting node, or by having a set of nodes that always remain infected, with a small probability that they will pass on their infection.

- When a virus infection is detected, it is good practice to warn others of the detected virus. These others then scan their machines looking for the virus, thus increasing the probability of detection and elimination and also decreasing the probability that these machines will become infected in the future. This can be modeled as a "kill signal" that propagates from a cured machine in much the same way the original virus propagated.

- An obvious extension to the model would be to allow machines to become immune to the virus. A further extension would be to allow machines to pass this immunity on to others, propagating in much the same manner that the "kill signal" discussed earlier.

- Incorporating some of the preceding ideas into a model with multiple viruses would be very interesting. In this model, a virus detection could not only warn others about that specific virus, but due to an enhanced vigilance the probability that other viruses would be detected would increase. This is the so-called "Michaelangelo" effect discussed in Kephart et al. [1993] and Kephart and White [1993].

- A better model for real networks would be a random graph where the graph itself changed with time. One way to model this would be to have time-

varying β_{jk}, allowing values to be zero occasionally, indicating no connection. Also, the size of the graph changes. Nodes are inserted (new machines coming online) and deleted (old machines taken off the network). This kind of dynamic network model introduces many interesting areas for future research.

One issue not discussed in this work is that of scale. While Equation (6.4) can be solved fairly easily for networks with hundreds of nodes, to model the spread of viruses on the Internet, networks will have to contain millions of nodes. It may not be a trivial matter to scale these calculations up to networks of this size, particularly if some of the modifications discussed earlier are implemented. However, it may be possible to do asymptotic analysis under the assumption that networks with millions of nodes are approaching the asymptotic regime. Investigations of these issues might be fruitful.

6.5 IMMUNOLOGY

The team at the University of New Mexico has adapted its immunology-inspired approach to the detection of viruses. Recall from Section 5.4 that the basic idea is to generate a collection of "n-grams" which correspond to "self" and to use the number of "nonself" mismatches to determine the likelihood that an anomaly has occurred. The analogy is taken one step further in D'haeseleer et al. [1997]. A biological immune system has antigens that are specific to certain pathogens. These antigens actively detect "nonself" by matching parts of known pathogens.

In order to implement something such as this, one could take the approach of most of the virus scanners: make specific detectors for each known virus. The problem with this is that it makes the detection of new viruses difficult.

The approach taken in D'haeseleer et al. [1997] is a little bit different. The idea is to generate a large collection of n-grams designed to detect "nonself." These are analogous to biological antigens.

D'haeseleer et al. [1997] provide three algorithms for generating these "nonself" n-grams. The first, referred to as the generate-and-test algorithm, is to generate a large number at random, test them against "self," and throw away those that match. The remaining strings, by definition, match "nonself." This is the simplest of the three methods. The others, which require specific knowledge of the matching algorithm employed, will not be discussed here. The interested reader is encouraged to read D'haeseleer et al. [1997].

The D'haeseleer et al. [1997] approach, in its purest form, does not require any examples of viruses. Obviously, if such examples are available, it makes sense to use them. This is a part of the architecture discussed in Marmelstein et al. [1998]. The idea is actually more sophisticated than simply using a few captive viruses to generate virus-detector n-grams. Instead, the idea of a "decoy" file is introduced (this idea appears to have originated in Kephart [1994] and Kephart et al. [1997]). A decoy file is one designed to be infected by viruses. These files are never used by the computer and by design should never change. Their only purpose is to exist, awaiting infection by viruses. Once an infection has occurred (detected via

detecting a change to the decoy file), the virus can be the source of "nonself" patterns.

Obviously, the selection of "decoy" files is critical to this process. In order to increase the probability of infection, a genetic algorithm is used to try to find files with attributes (filenames, location, size, priority, etc.) such that the probability of infection is maximized.

An alternate approach, not discussed in the preceding work, would be to try to estimate the probability of infection for a specific set of attributes. This estimation would require the modeling of the set of all possible attributes and the collection of a large amount of data, presumably via repeated infection of various machines by a wide range of viruses. This approach appears infeasible when stated in these terms, but it is in effect what the genetic algorithm is attempting to do in Marmelstein et al. [1998].

In a series of papers (Hofmeyr and Forrest [1999], Hofmeyr and Forrest [2000], and Forrest and Hofmeyr [In press]) Steven Hofmeyr and Stephanie Forrest detail an artificial immune system for computers. This can be used for any kind of intrusion detection, whether network-based or host-based, unauthorized use, or virus detection. As before, the basic idea is to construct a collection of detectors. These detectors are bit strings, which are matched against "foreign material" (code or packets) that enters the system. They use the "r-contiguous bits" matching rule: two strings match if they have at least r identical, contiguous, bits. These bits need not be in the same position within the string but must be contiguous. One could extend this rule, for instance, to allow at most q mismatches within the r bits.

The construction of the "nonself" detectors is closely related to biological immunology. The interested reader should consult the papers cited, where this relationship is discussed in considerably more detail than we have room for here. The basic idea is as follows. A detector is created at random and labeled "immature." If a detector detects a match while it is immature, it is deleted. The idea is that "nonself" is rare, so anything an immature detector finds will actually be "self." Once a detector "matures" (after a fixed amount of time), it becomes part of the database of "things that should not be detected." This is analogous to lymphocytes, which look for "nonself" to bind to and kill.

In addition, the system can be trained with "nonself" to construct detectors specifically designed to detect certain known attacks. In addition, when a detection does occur, a "nonself" detector that matches the attack exactly can be constructed to speed up future detections.

The system is also adaptive. This is implemented by giving the detectors a life span. Each detector has a probability of dying (disappearing from the database). In this manner, as the system changes, the detectors can adapt to the new "self," and old detectors that are no longer valid do not clutter up the system.

This work is extremely interesting and shows great promise. It remains to be seen whether it can be made practical, but the results to date are quite impressive. Even if it turns out to be impractical for individual systems, it should lead to very interesting immunological models, which may be of interest in their own right.

In a very short paper, Gilfix [1999] discusses extending the immune system idea to overall system and network management. This is little more than a proposal at this point.

6.6 VIRUS PHYLOGENIES

As with biological viruses, there is an interest in tracing the "ancestors" of computer viruses. This section will discuss some work on constructing these phylogenies.

Unlike biological viruses, it is possible for a computer virus to be created with no ancestors. Also, except for polymorphic or other evolving virus types, computer viruses don't actually "give birth" to new species. These differences are not that important, really, but are worth keeping in mind while we look at the work.

The main references for this work are Goldberg et al. [1991], which documents the original work, and the later journal article documenting this work, Goldberg et al. [1998]. The basic idea is that new computer viruses are often written using ideas, or even code fragments, from previous viruses. By matching the code strings in these viruses along with the dates of first detection of the viruses, one can produce a phylogenic mapping of the ancestors and descendants of the viruses. Some care must be taken to ensure that the code strings used are long enough to be valid. The analogy to keep in mind is biological phylogenies based on genetic maps. Obviously, these genetic maps must be made using sufficiently long segments of DNA (for example, genes). A phylogeny based on looking at segments of only a few base pairs would be useless. Similarly, in the computer virus case, there are certain operations that must be performed by computer viruses, such as file reads and writes and memory copies, and so on. Only relatively long sequences of code will be of value in constructing meaningful phylogenies. Goldberg et al. [1998] suggest using sequences of 20 bytes or more.

The basic technology used in constructing virus phylogenies is to compare the binary code of the viruses, finding sequences that are common to all of those in the collection, and then computing the probability that a sequence of that length would occur by chance in such a collection. If the probability is small, then the viruses are assumed to be related, with a common ancestor (which may be one of the viruses in the collection).

The assumption is made that (for sufficiently long code fragments) each fragment is invented only once. This is a reasonable assumption, assuming the fragment is "long enough," an assessment that is to a degree subjective. This assumption can be tested. Collect a large number of programs for which there is no reasonable chance that any of the code of one program was borrowed from another, and test to see whether any code fragments of a given size are found in more than one program. If some are found, one has an estimate for the probability that such fragments would be found by chance. This probability can then be used to give a confidence for the constructed phylogeny.

A phylogeny is defined to be a directed acyclic graph. Recall that a graph without cycles is called acyclic, while the adjective "directed" means that the edges have a direction associated with them, and that hence any cycles must follow the direction of the edges.

By comparing bit strings within the virus code, a phylogeny is constructed showing the implied relationships among the different viruses. In this way, one can determine when particular ideas first appeared, and which viruses built on ideas from previous viruses.

The main contribution of Goldberg et al. [1998] is an algorithm for constructing these phylogenies. They provide a fast (greedy) algorithm, and provide a proof of its performance. Like most greedy algorithms, it is not guaranteed to find the optimum match, but it is off by at most a factor of (approximately) the log of the input length.

Further work on the actual construction of phylogenies for specific collections of viruses is needed. This would be particularly interesting for the "new" macro viruses (see Sections 6.7.3.1 and 6.7.3.2). My guess is that they owe quite a lot to earlier viruses.

A more ambitious but related effort is proposed in Spafford and Weeber [1993]. This proposes looking at the source code with the purpose of determining the author of the code rather than constructing a phylogeny. This would make use of choices of data structures, coding and formatting styles, library calls, grammar and spelling errors, and so forth. This is a good place to utilize statistical techniques, as are used in the determination of authorship for written documents.

6.7 WORMS

6.7.1 Introduction

A worm is a program that, like a virus, reproduces itself. The distinction is that a worm makes new running copies of itself instead of infecting files.

The world's simplest worm (Do NOT try this at home! Or anywhere else!) is

```
#!/bin/csh
echo "Wiggle Wiggle"
$0 &
$0 &
```

The first line indicates that this program is to be run under the "cshell." The "echo" is simply to give the worm some effect besides propagation. This would be replaced by any action that the worm is expected to take prior to propagation. The "$0" gets expanded by the shell to the name of the calling program, in this case the name of the file. The "&" puts the program in the background.

This particular worm is also called a "fork bomb" because it forks off processes, which in turn fork more processes. Some people would not consider this kind of program a worm since it does not attempt to propagate itself outside the originating computer.

If executed on a Unix machine, this will print "Wiggle Wiggle" to the terminal, then spawn two copies of itself, each of which will print and spawn two copies. Even though the individual processes exist for a tiny amount of time, this quickly fills up the processes table and the computer slows to a crawl (or crashes). The only reliable way I have found to shut the thing down is to reboot the computer. It may be possible to kill it with the command:

```
kill -9 -1
```

which will kill all processes owned by the user that executes this. If you are root, it will bring the machine down. However, if you execute it as a normal user, there is no guarantee that it will work on the "wiggle" worm.

A "safe" version of this program is as follows:

```
#!/bin/csh
echo "Wiggle Wiggle"
$0
```

This one is relatively safe to run. It will print a bunch of "Wiggle Wiggle"s to the screen until it is killed (via a control-c). At any time only one version of the program is active. I say "relatively safe" because even this program will eventually suck up all the resources on your computer. It is a recursive function with no exit criterion, and each call gets a new process ID. The program is easy to kill (unlike the previous worm) and so is really not much more dangerous than an infinite loop in a program. However, it can, if left unchecked, bring a machine to its knees, so do not try it on any machine that is not your personal property.

Here is another twist on the "wiggle" worm. This one renames itself:

```
#!/bin/csh
echo "Wiggle Wiggle"
set d=`date`
set n=`echo $d[4] | sed "s/://g" `
@ n *= 131
@ n %= 10000
@ n *= 71
@ n %= 40000
set name=`head -$n /usr/dict/words | tail -1`
mv $0 $name
sleep 1
$name
```

The single quotes in this program are "backquotes." These cause the enclosed command to be executed. The variable is set to value that the command returns. The lines starting with "set" and "@" are intended to select a random name from the local dictionary (this is not a particularly good pseudo random number generator, but it suffices to make the point). This program will start one copy a second, each copy showing up in the process list with another name. It gets these names from the local dictionary, /usr/dict/words. This is certainly not the most elegant such program, but it will work on most Unix systems, provided the dictionary file is changed to match the local system.

Running the preceding script for a few seconds produced the names

Dewitt Esmark acclimates atrophy brassy cocking gasser grime inherits loafed oblivion polygons regaining sensitively

Note that the "random number generator" is far from random. Obviously, a better random name generator (which did not use "date" and therefore could start more than one worm a second) would be easy enough to write.

Change the "$name" to "$name &; $name &" in the program and you will very quickly crash your system. Once again, I must beseech you: DO NOT DO THIS. It is merely for the purpose of illustration. Playing around with malicious code may very well get you fired, fined, or jailed. The writers of the Internet worm and the Melissa virus were not treated as harmless pranksters when they were caught. It is a very serious offense and will be prosecuted if anything goes wrong.

Unlike their biological counterparts, a worm can be a virus and a virus can be a worm. We will see several examples of worms that are also viruses. Some of the more famous "viruses" were both viruses and worms.

Are worms bad? After all, many important problems require vast computing resources, and most home computers are sitting idle most of the time. Why not use them? This is the approach taken by the SETI (Search for Extra Terrestrial Intelligence) people. The idea is to provide a screen saver that processes a piece of the vast amount of data that the SETI program has collected. This distributes the computation across a very large number of machines (as of January 15, 2001, the SETI Web page reports over 2.5 million users, for a total of over half a million years of CPU time). The key is that this is a purely voluntary activity. Anyone who wishes to participate is welcome to obtain the software and install it on their computer. If they decide they no longer want to participate, they simply stop using the screen saver. At no time is the program propagated to another machine without the permission and knowledge of the owner of the machine. See

http://setiathome.ssl.berkeley.edu/

for more information about this project.

The distinction between this kind of voluntary distributed computing and worms is clear. A worm or virus infects a machine by obtaining unauthorized access. Although it is true that most machines are idle most of the time, this does not give one license to use these machines without permission. It is as if one decided that since people don't use their cars at night after they go to bed, it should be okay to borrow them during those hours.

Detecting worms is generally quite easy since they tend to be quite greedy, using up resources quickly and moving rapidly from machine to machine. This is an artifact of the way previous worms have been written, however, not a fundamental property of computer worms. Assuming that the purpose of a worm is not simply a denial-of-service attack, the only reason for it to replicate quickly is to avoid destruction. This is only necessary if the worm is detected. Thus, the future may hold very subtle worms that spread slowly, gathering information and waiting for a certain stimulus before making themselves known. In some sense, from a benign point of view, this is what some people have envisioned as the way intelligent agents are destined to be written.

6.7.2 Internet Worm

On November 2, 1988, the Internet was infected by a worm (Rochlis and Eichin [1989], Spafford [1989], Denning [1990c]), which became known as the Internet worm. The Internet worm, written by Robert Morris, a computer science graduate student at Cornell University (Eisenberg et al. [1989]), exploited a hole in the sendmail program that allowed it to propagate itself from machine to machine. Once a machine was infected, the worm spawned more and more copies of itself until the machine was so bogged down that either it crashed or was pretty much useless to anyone and had to be rebooted.

The worm infected between 2000 and 6000 machines, which corresponded to between 3% and 10% of the machines on the Internet at the time (Denning [1999]).

The worm was actually quite sophisticated. Once it infected a host, it set about trying to find new hosts to attack. It did this in various ways, such as by looking in the mail forwarding files of the users on the machine and looking at host tables for trusted hosts. Simultaneously, it set out to crack the password file on its current host. This was done by first trying a collection of common passwords (McAfee and Haynes [1989] pp. 89–90) that the worm brought with it, and if this did not work (it often did) trying the dictionary resident on the compromised machine.

The worm spread by utilizing an exploit in sendmail, by trying to log in as one of the users it had discovered, and by trying an exploit against the finger program. Once it obtained access to a new machine, it sent a bootstrap program, compiled it, started it, and closed the connection. If all went well, the bootstrap program called the original machine back (actually, it made a tcp connection), and the parent worm sent across the rest of the code.

The program also hid itself on the compromised machine. It removed its files from the disk, running in memory only, and changed its process name to look innocuous.

As stated earlier, the worm was actually fairly innocuous. It did not use too many resources, was not overly destructive, and would disappear from a system upon a reboot. This does not make it a good thing, but if that were all it did it would not have been the disaster that in fact it was. The problem was that it propagated faster than it died. A newly infecting worm would look for copies already on the machine, and if found, some copies would be killed off. Unfortunately, not all copies were killed, and they propagated to many hosts before they died. Had Morris written the worm so that it would never have more than one copy on any host and the copy would terminate after a short period of time, the Internet worm might even have gone undetected after (briefly) infecting nearly every machine on the Internet. It did not work out that way.

6.7.3 Macro Worms

There was a hoax which popped up periodically about a virus that was being sent by email. There were several variants, but the gist was that if you read a particular email message, your computer would be infected with a virus. The computer literate among us would laugh at this since, after all, everyone knew you couldn't get a virus by reading email. We are not laughing now. (Well, maybe a little.)

It should be pointed out that we were not wrong. You cannot get a virus by reading email. You never could, and you can't now. However, email readers no longer simply display the text for you to read. Helpful software vendors have added nice features to allow people to send documents and programs and to have the email reader interpret (run) the programs so that the reader gets the full effect of the message. Thus, we are no longer reading email, but rather running it.

I recently received a hoax email like the one mentioned above. This one informed me that a new virus had been detected that destroyed one's hard drive. It looked like a real warning, complete with links to news stories (which turned out to go to the home pages of the news organization, not to a specific story). At the end of the message as a note telling me to forward the message to all my friends.

This last line caused me to re-evaluate my impression of the email. Instead of considering it a hoax, I now view this as a worm itself. By forwarding the email on, I would be propagating the worm. I find this view intriguing. In a sense, this is a worm that is never executed (except in a person's brain, if you will), and yet can propagate via the Internet.

Macro worms are usually referred to as macro viruses, primarily because the term "virus" has caught on with the public much more than the term "worm." It is perhaps a bit pedantic to call these worms, particularly since many of them also act as viruses. The ones that have made the most press are still definitely worms by our definition.

The term "macro" comes from the old idea of being able to program single keystrokes to perform multiple tasks in one's word processing software. These were called macros, and were extremely useful, allowing users to customize their systems so that commonly used sequences could be performed with a single keystroke.

Not willing to let well enough alone, the writers of word processors and other software systems made these languages more and more powerful. This is always good from the perspective of the users since it makes it even easier to do more wonderful things. It would still be good, if either security had been a strong focus of the developers or the software industry had not gone more and more into bundled software.

Now, when you read your email, if it contains a document in a word processor format, your reader will display the document for you, calling the appropriate word processor seamlessly. This is very good from your perspective since you don't have to worry about formats, saving things to disk, finding the right program to read them, and so on. Unfortunately, the document also executes all the appropriate macros, which is where the trouble starts.

In the following sections, we will look at two of the most famous (at the time of this writing) macro worms.

6.7.3.1 Melissa Friday, March 26, 1999, was an interesting day for computer security professionals. (Remember the famous curse: May you live in interesting times.) This was the day that many people (including some security professionals) found in their mailboxes a message containing a document, which, when opened, spread a worm (which was also a virus) to their colleagues. This was the debut of

Melissa, and it literally changed the way people thought about viruses, email, and the Internet.

The Melissa virus took advantage of two useful functions that modern mail systems provide for their users. The first was discussed earlier. Mail readers allow the user to mail documents (as attachments), and these documents can be read by the recipient without the hassle of first saving them, exiting the news reader, finding the right application, and so on. One merely "clicks" on the attachment, and the document is displayed. With the document can come code (macros) that is also executed when the document is opened.

The second feature that macro worms such as Melissa use to great effect is electronic address books bundled with the email software. This is really an essential part of any email system, which allows one to associate the person's name with the email address. Without address books, email systems would simply be too awkward to use.

Melissa was an email message that had the subject line: "Important Message From" and the name of someone you know. It came from someone you know, and was addressed to you, not some email list you belonged to. In the body of the message was the phrase "Here is that document you asked for" followed by a (Microsoft Word) document. When you opened the document, it looked in your address book and sent itself to everyone in it (at least, to the first 50 addresses in the list). Thus, they received "personalized" email from you (their friend), and so it went.

It went one step further. It infected your Microsoft Word software so that new documents you created would be infected. This is why Melissa was in fact both a worm and a virus. It also changed the security settings on your Microsoft Word software to make your system more vulnerable to macro viruses and harder for you to increase the security level.

Melissa cost millions of dollars (Garber [1999]) in lost time, services, and an unknown amount of productivity caused by everyone talking about the virus instead of doing any work. I know of several large facilities where the network connections to the Internet were closed down for more than a day while the process of recovering from the virus and cleaning the infected machines was performed.

Tracing Melissa to its author might have been all but impossible except for one oversight. Microsoft Office 97 puts hidden data in its documents identifying the machine on which the document was created. This is how the perpetrator of Melissa was finally identified. Of course, future authors of malicious code will simply change these bytes to cover their tracks.

6.7.3.2 I Love You The "I Love You" virus (ILY) hit on May 4, 2000. Another version of a macro worm that attacks Microsoft platforms, it was the next stage of evolution from Melissa. Its name came from the subject line: "I love you," a message that is hard to resist, particularly coming from someone you know.

ILY was truly both a worm and a virus. Like Melissa, it spread by sending copies of itself to people in your address book. It also changed the default Web page for Microsoft Internet Explorer to connect to a page that executed the virus. It added files to the computer so that anyone who connected via Internet Relay

Chat (IRC) would become infected. Finally, it changed image and music files so that they executed the virus.

If this was all that it did, it would be very bad. But ILY went one step further. It mailed dial-up account names and passwords to a site in the Philippines, so that the author of ILY could use these accounts for free.

When ILY modified an image or music file (say a JPEG image), it made it an executable (VBScript) file. This is accomplished by changing the content of the file, and making the extension ".vbs" instead of ".jpg." People used to Unix might wonder how such a thing could go unnoticed. How could you execute an image by accident? But remember, Microsoft Windows has the philosophy that files should be tied to the application that opens them, and when you select a file, you should run the appropriate application on it. Thus, when someone wants to view an image or listen to a tune, they simply click on the icon or name associated with the file, and the operating system takes care of running the appropriate program. A further aid to ILY is the convention that files do not display their extensions (unless the user specifically wants to see them), so the change in file name generally goes undetected.

Like Melissa, ILY caused a lot of consternation, made big headlines, and cost a lot of organizations a lot of money cleaning their systems. It probably also made a lot of money for companies that detect and eliminate viruses.

6.7.4 Ramen

In early January, 2001, a new worm was detected that compromised Linux systems. This worm uses a script of attack tools to first compromise new systems and then install itself on the new systems for further propagation.

This worm illustrates the sophistication of recent malicious code. Like the distributed denial-of-service tools (Section 7.5.1), it utilizes several different attacks, depending on the vulnerabilities found on the system. It looks for FTP servers, particularly a version of wu-ftp with known vulnerabilities, rpc.statd, and LPRng. It is reported to be easy to add new exploits to the worm due to its scripted nature.

Once the worm has found a vulnerable system, it installs itself on the system, setting up a Web-like server on port 27374. It provides a copy of itself to any request on that port. It also searches the disk for files named "index.html" and replaces them with its own page. It sends email to announce the compromise of the system. This could, potentially, provide a mechanism to track down the originator of the worm, although with free email and public terminals it is unlikely the perpetrators will be caught this way.

Finally, it scans for new machines, scanning for services on port 21.

Thus, ramen can be detected via a variety of ways. First, any port scans to port 21 should be investigated. If your network does allow FTP activity through the firewall, an anomaly detection system that looks for connection requests to machines that do not run FTP, scans, and connections from machines that do not normally access the network may be able to detect the initial attempt to propagate the worm. Any outgoing or internal FTP scans should be investigated as potential evidence that a machine has been compromised (these should be investigated as a matter of course since they are an indication that something bad is happening). The

security officer should add port 27374 to the list of ports scanned by vulnerability assessment software such as nmap or saint.

Obviously, the first order of business is to patch the systems with the vulnerabilities. This should always be done upon the announcement of a new vulnerability or software that exploits a known vulnerability. Since programs such as ramen are easily adaptable, the indicators listed earlier (FTP scans, port 27374) can be modified in future versions, so the ultimate detector must look for unusual behavior of all types to detect the new attacks for which prior knowledge of their signatures is unavailable.

6.7.5 Statistics and Worms

We close out this chapter with some thoughts about detecting worms. One approach worth considering is that described in Section 4.8. The GrIDS system might be used as a worm detector by constructing graphs showing connections between nodes. A large tree or, in the case of a worm such as the email worms described in Sections 6.7.3.1 and 6.7.3.2, a graph with high degree (number of edges per vertex), might be indicative of a worm.

A single machine sending email to a large number of recipients should be a tipoff of a potential problem. This is only true if the machine does not do this on a regular basis. For example, a machine that maintains an email list might regularly send copies of a single message to a large number of recipients. Most desktop systems will do this very rarely, if at all.

Similarly, one could look at the process table and look for a large number of processes or for many short-lived processes. One could monitor the number of processes, the amount of memory, load average, or other measures of system performance and flag unusual deviations from the normal range. For many systems, this will give a good early warning of problems. On my machine, I regularly use nearly all of the available memory, but rarely do I spawn a large number of processes. Therefore, just as in Section 4.5.4 where we modeled "normal" behavior for network traffic, one needs to do the same thing for system performance monitoring.

Some of the visualization techniques discussed in previous chapters are of value in the detection of worms. For example, the data image can be used to investigate the progress of a worm by plotting source IP against destination IP, as was done in Shoch and Hupp [1990]. A plot of IP address against load average could be used to detect a CPU-intensive worm as it spreads through a network.

6.8 FURTHER READING

There are a number of articles on computer viruses for the lay person. Denning [1990a] is a nice short piece, supplemented in the same book by Spafford et al. [1990]. Ashmanov and Kasperskaya [1999] describes a virus encyclopedia available at

http://www.viruslist.com.

Hedberg [1996] describes the work done at IBM, including Kephart's work on computer immunology. Peláez and Bowles [1991] give a classification of "malicious code" such as viruses and discuss the various kinds of "beasties" that have been developed.

Cohen [1987] and Cohen [1991] are good places to start learning about computer viruses, as is the "Random Bits & Bytes" column of Harold Highland (see, for example, Highland [1988] and Highland [1989]). Hruska [1997] gives a brief description of virus scanners, while Kensey [1993] discusses several issues involving computer viruses.

For the mathematically inclined, Andersson [1998] provides some limit theorems describing the time evolution of random graph models of virus epidemics. In this work, an SIR epidemic is considered, where a homogeneous population is assumed but the individuals have a fixed number of acquaintances. There are a number of similar papers describing different modifications to the basic assumptions of the epidemic. These include Nåsell [1999], which considers the time to extinction for a class of epidemic models, and Andersson and Britton [1998], which looks at modeling an epidemic among a population that has varying susceptibility to the diseases.

The book by Andersson and Britton [2000] gives a nice introduction to these issues and stochastic modeling of epidemics in general. The review article Hethcote [2000] covers the basic models and discusses their applications to human diseases. Like most Siam Review articles, this is quite accessible to the non-expert, while being quite thorough. It also has an extensive bibliography. Another good reference is Daley and Gani [2000].

Another approach to virus detection is discussed in Lee et al. [1997]. This incorporates an emulator, which provides a simulated environment in which the virus can be safely executed, and an analyzer, which does the detection and analysis. Lo et al. [1991] discuss an architecture for a testbed to detect malicious code and give a brief taxonomy of malicious code.

Tesauro et al. [1996] report on a neural network for the recognition of computer viruses but without enough details to evaluate the technique.

A very interesting twist on the computer virus is described in Young and Yung [1996]. The idea is that rather than destroying files, viruses could utilize public key cryptography to encrypt files. The virus writer could then offer, for a fee, to decrypt the files. I am not aware of any cases where this form of extortion was attempted, but if it were successful it is unlikely that the victim would publicize it.

7

Trojan Programs and Covert Channels

7.1 INTRODUCTION

We are all familiar with the story of the Trojan Horse. The Greeks built a large wooden horse (or rabbit, according to Monty Python), rolled the horse up to the gates of Troy, and left. The Trojans, thinking this was a gift, brought the horse inside the gates. Unbeknownst to them, the horse contained Greek warriors, who sneaked out under cloak of darkness and opened the gates, letting in the rest of the Greek army, resulting in the sacking of Troy.

The Trojan Horse was something other than it appeared. In the same sense trojan programs are ones that are not what they appear. They come in all shapes and sizes. Some simply replace existing programs with ones that perform additional (and undesirable) functions. For example, one might replace the "telnet" function with one that is identical to telnet with the single addition that the user name and password are retained and saved somewhere for future pickup. Others masquerade as useful or amusing utilities, which when executed open up "back doors", allowing access to the machine by anyone who knows how to utilize the back door. Some are simply one-shot programs: upon execution they do something really nasty, such as reformat the hard drive. Others stay dormant, awaiting some external event to activate them.

A trojan was installed on a machine at George Mason University in Fairfax, Virginia, in the Netscape browser (Denning [1999], page 261). Whenever anyone brought Netscape up, a protest email message was sent to the local security review panel. After a number of students complained about receiving email replies to messages they had not (to their knowledge) sent, the problem was tracked down and the trojan discovered.

Like viruses and worms, it is not a simple matter to distinguish a trojan from another of these denizens. A trojan can replicate itself and thus be either a virus or a worm, or both. Some programs nevertheless clearly fall into the category of "trojan," so in this chapter we will consider some of the characteristics of trojan programs.

In this chapter, we will also consider covert channels, one of the mechanisms that trojans use for hidden communications. These are of interest in their own right and are of critical importance in the area of multilevel security. We then briefly consider steganography, which concerns hiding messages in other messages. Then, we will look at a few common trojans and consider methods for detecting trojans.

7.2 COVERT CHANNELS

A covert channel is a communication channel that is hidden or otherwise not apparent to others. For example, imagine you and a partner are planning the Great Bubble Gum Robbery of 2003. Your partner is currently in temporary seclusion (to be paroled in another month) as a result of the foiled Great Lottery Ticket Scam of 1996. You need to communicate through letters without worrying about interception by the authorities. You know that your letters are read by the prison officials, so you institute the following scheme: you both have access to computers, so you will use a different font for the real message. Thus,

"Your mom is *th*rilled about your coming release. An*g*ela *u*rges you to rem*a*in a model p*r*isoner. *D*oes *ch*al*l*en*g*e of *a* legi*ti*mate l*i*festyle excite you? We know you'll *do no*thi*ng th*at will jeopardize your freedom."

sends the real message: "The guards change at midnight." This is a covert channel, a communication hidden within another overt communication channel.

The preceding example is not a particularly good covert channel since it is easy to detect. Denning [1999] provides a similar example. The message is encoded as the first letter of each word in the following cable:

"President's embargo ruling should have immediate notice. Grave situation affecting international law, statement foreshadows ruin of many neutrals. Yellow journals unifying national excitement immensely."

As you might imagine, it is not an easy task to construct such an encoding in a manner that does not arouse suspicion. The typically stilted phrasing of cables may help here, but it is still not a trivial matter.

Other approaches are easy to construct, particularly if one moves from the somewhat cumbersome arena of human communication to computer communication. We will look at a number of ideas for implementations of covert communication channels and consider methods of detecting them. Some simple ideas are:

- A Web server sends packets containing the contents of the Web page to any machine that connects and requests the page. The Web server controls the size of the packets sent to specific machines, encoding the covert channel as the packet sizes.

- The IP and TCP headers have a number of fields whose values are not specified or are unused in some cases. For instance, the urgent pointer is only used if the urgent flag is set. This field can be used to send covert messages. The program covert_tcp implements a covert channel using sequence numbers to encode the message. More information on this (including the covert_tcp program) can be obtained at

  http://www.firstmonday.dk/issues/issue2_5/rowland/

- A company decides to disallow telnet sessions into their facility and institutes a firewall policy to deny them. A user bypasses the firewall by using the data field in ICMP packets to implement a covert telnet. This is essentially what the Loki program does.

A good place to start learning about covert channels is the technical report NCSC [1993]. A brief (two-page) discussion on covert channels is found in Millen [1999]. A very entertaining article on covert channels is Simmons [1998a].

Covert channels are a serious problem for multilevel security systems. Consider a system with two security levels, Low and High. In a military situation, these might be different levels of classifications, for example unclassified and secret. Low can write to (send information to) High, but High cannot write to Low. In the classification analogy, unclassified information is allowed to pass to systems cleared for secret material, but secret information must not be transferred to an unclassified system. If a covert channel can be implemented, however, the security can be breached.

Consider the case where the High system must be able to ACK data from the Low system. Sending the ACK directly is obviously not acceptable, even if it is constrained to be a single bit (1 for ACK). A simple covert channel can be set up via timings of the ACK responses.

In Moskowitz and Kang [1994b], a statistical communication channel is defined. The original definition was defined in terms of response times, but we broaden the definition slightly here:

Definition 1 *If High can affect a parameter of the distribution of some system attribute measurable by Low, we say that there is a* **statistical channel** *between High and Low.*

Thus, there must be some entity between the High and Low systems that acts as a mediator. This mediator is called a "pump." A detailed discussion of a network pump can be found in Kang et al. [1996]. The basic idea is quite simple. Place a buffer between High and Low. The buffer takes the ACKs from High, and releases them to Low with times chosen from a given distribution. This can be made more reliable by using multiple buffers, with buffers for the data from Low, as well as buffers for the ACKs from High.

The pump must be stateful in the sense that it needs to remember the sequence numbers of the packets to ensure that High does not try to ACK packets that it did not receive (and hence construct a channel either using bad packets as bits, or using

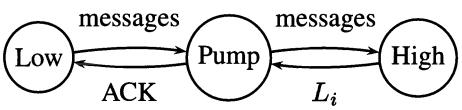

Fig. 7.1 A simple pump. The pump buffers the ACKs from High and passes them down to Low at a random rate L_i.

the sequence number to transfer data). All the fields sent with the ACK (usually only a sequence number and ACK flag in these systems) must be prescribed to ensure that data are not sent in unused fields.

The goal is not to eliminate covert channels but rather to reduce their bandwidth to as near zero as possible. It is always possible to construct a covert channel through manipulating the statistics of the responses, but with care this can be made to have a very small bandwidth. Further, the more difficult the channel is to construct, the more likely it is that it will be detectable (at least, one hopes so).

As an example, let us consider the simplest pump. A pump consists of a buffer of size n (we will assume n is large enough that it is never full). Messages are passed from Low to the pump and then on to High. The ACK from High goes to the pump and into the buffer. Let $\bar{H}_m$ be the average of the past m High response times. The pump releases ACKs from its buffer at a rate L_i. The rate L_i has a density that is a function of $\bar{H}_m$. This is illustrated in Figure 7.1.

In this simple example, High can instantiate a covert channel by manipulating its response rate to change the statistical distribution of L_i. By using multiple buffers, this can be made more difficult. More information on these pumps can be found in Moskowitz and Kang [1994b], Moskowitz and Kang [1994a], Kang et al. [1995], and Kang et al. [1996]. Related techniques are described in Venkatraman and Newman-Wolfe [1993] and Browne [1994].

An interesting idea has been developed by Ronald Rivest at MIT. Called "chaffing and winnowing," the idea is as follows. Suppose John wants to send a message to Mary, but does not want anyone else to be able to determine what the message said. One thing John and Mary could do is decide on an encryption scheme. Rivest's method is a little different, since it allows the text to be sent unencrypted but still undecipherable by a listener. John and Mary decide on an authentication method. This is like the checksum that tells the TCP/IP stack that a packet has been delivered intact. For example, let us suppose that the authentication consists of the ascii value of the first letter in the message (obviously a more sophisticated authentication would be used in real life), so a message might look like

```
1:We need:127
2:to sell:164
3:all shares:141
4:of Consolidated:157
```

```
5:Widgets:127
6:immediately:151
```

The first number is a sequence number, so that the message can be put together properly, followed by a few characters of text, followed by the authentication number. Just like IP, we break the message up into small packets that get sent individually.

The clever trick is that John sends this message along with a set of fake messages (which will fail the authentication test). These fake messages are the "chaff" that confuses any listener. Without the proper authentication algorithm, the listener cannot tell which message fragments are authentic, and which are confusers, so the full session might look something like

```
1:We need:127
1:We must not:131
1:Your mother:117
1:There is a:117
2:buy:154
2:to sell:164
2:wants to:164
2:fine line:164
3:all shares:141
3:bake a:141
3:marry:141
4:of Consolidated:157
4:between:172
4:pie:152
4:perfume:152
5:Widgets:127
5:baker:127
6:immediately:151
6:tomorrow:151
6:and aftershave:151
```

Note that if the amount of chaff is large relative to the message, and the chaff comes from legitimate messages, it will be extremely difficult to extract the true message. Further, if John wants to send messages to Mary, Jane, and Esmerelda, he can use different authentication algorithms for each recipient, and the message to Mary becomes chaff to Jane and Esmerelda.

If the amount of chaff is relatively small and/or does not come from legitimate messages, it may be possible to extract the true message as the only combination that makes sense. In an extreme case, each packet could consist of a single character, which makes reconstructing the message from the chaff using this kind of textual analysis extremely difficult. If one adds encryption to the message prior to breaking it up into packets, then textual analysis will fail even in a "low-chaff" environment. The purpose of the technique was to argue that the desire of the U.S. Government to control encryption technology was misguided and futile, so adding encryption goes against the original intent of the work.

The paper describing this idea is available at

http://theory.lcs.mit.edu/~rivest/chaffing-980701.txt

One idea that I have not had a chance to test is that it may be possible to use this "chaff" idea to fool some content-based network monitors. Any such system must put the packets together in order to look for suspicious strings, particularly in applications such as telnet, where each packet may contain a single character. If one were to add in a few packets in the right places, with invalid TCP checksums it may be that the monitor would use their data in the reconstructed content. This assumes that the monitors do not check the checksums and that the fact that there would be multiple packets with the same sequence number would not cause an alarm by itself. This is something for the developers of such systems to consider.

Alternatively, one could install a trojan that used a sniffer and only used packets that failed the checksum and were hence discarded. The data in the discarded packets could be used for the channel. If the intrusion detection system does "correctly" reassemble packets, ignoring those that do not pass the checksum test, they would miss these. As you can see, it is very hard to take every eventuality into account. It would be interesting to know whether any network monitors check for packets with bad checksums.

7.3 STEGANOGRAPHY

Steganography ("covered writing") is the art of hiding messages. We saw an example of this in Section 7.2 when the two would-be masterminds used different fonts to hide their messages. Another famous example is the use of micro dots hidden in the periods of letters. Herodotus (Herodotus [1998] 5:35) tells us that Histiæus shaved the head of a slave and tattooed a message on it. Once the hair had grown back, it covered the message. Since the message was to tell Aristigoras to rebel against the king, it was important that only Aristigoras read it. The slave was sent to Aristigoras, and the message was delivered. This and other such stories can be found in the introductory chapter of Petitcolas [2000] and in Jamil [1999]. An overview of steganography can be found in Johnson and Jajodia [1998].

Another use of steganography is digital watermarking. This is a way of marking (usually in a manner that is not readily detectable) images and other digital media in order to prove ownership or origination of the material. For example, if you are a photographer, you might like to display your images on the Web, but if someone uses your images without your permission, you may want to be able to prove that the image was in fact yours. This is not unlike branding cows except that it is designed to be less obvious and less painful to the cows.

Let us focus on hiding messages in images since images are a common target for steganography, and they allow for simple illustration. The first method that one might consider for hiding a message in an image is as follows. Recall that an image is an array of 8-bit (or if it is a color image 24-bit) entries called pixels. Let us just consider 8-bit, or grayscale, images. The value of each pixel determines its gray level, with 0 being black and 255 being white. As a result of the way

the human visual system is designed, we do not notice very slight differences in grayscale values, so first set all the lowest-order bits in the image to zero. Then, take your message, represent it as a bit stream, and for each bit in the message set the lowest bit of a pixel to that value. You can do this systematically or randomly, provided that you retain the seed to the random number generator so that you can extract your message and that you take care not to reuse any pixels. The image (or other digital medium) in which your message is hidden is called the cover.

Surprisingly enough, we need not be as stingy about our pixels as to use only the lowest bit. Figure 7.2 shows an example where a whole image is hidden in the lower 4 bits of another image. The procedure is to take the four highest-order bits of the image to be hidden and set the low-order bits of the cover to these values.

As can be seen from the figure, one cannot see the hidden image in the cover. There is a perceptible change to the cover, as can be seen in a higher resolution view, but this is really only noticeable if one has the original for comparison.

This form of steganography is extremely simple, and as a result, generally pretty easy to detect. Further, it can be destroyed by very simple image manipulation. For example, lossy compression will generally destroy the hidden image. This is shown in Figure 7.3, where we have extracted the hidden picture after undergoing JPEG compression with a quality setting of 75%. This resulted in a compression factor of about 7. Although the cover is not noticeably changed by the compression, the hidden image is nearly destroyed.

An approach to steganography that is more robust to compression and other filters applied to the image is to perform the embedding in a transform space, such as the Fourier domain. More information about this and other techniques can be found in Marvel and Retter [1998], Johnson and Katzenbeisser [2000], and Lee and Chen [2000]. This is by no means a complete list. The book by Katzenbeisser and Petitcolas [2000] is a good place to start learning about the subject, and the chapters therein contain an extensive bibliography.

Digital watermarking is slightly different in intent than steganography, but many of the same techniques can be used. It is important for commercial reasons to be able to mark an image, movie, or music with some kind of secure tag to detect or prevent unauthorized copies. However, as reported in Seife [2000], these techniques are not yet perfect. There is some controversy, but the gist of the article is that some researchers claim to be able to crack a wide range of existing watermarking technologies, essentially rendering them useless. Clearly, there is work to be done here (assuming you accept the need for this kind of technology).

We have seen another method for hiding messages in Section 7.2, where a message is passed by changing the statistics of a signal.

Basically, all steganography techniques come down to the following procedure. First, transform the cover using some transformation that leaves redundant bits. Select a (usually random) subset of the bits and tweak them to embed the message. Note that when computer scientists say "random" they almost always mean "pseudo-random." It must be possible to reproduce the pseudo-random sequence in order to extract the image.

One then (usually) applies the inverse transform to obtain something indistinguishable (to the casual observer) from the original but now containing the hidden message. Most of the work in steganography and digital watermarking comes

Fig. 7.2 Hiding dogs in cars. The top two images are the original. The bottom image contains the first image (rotated 90 degrees counterclockwise) in the lower four bits.

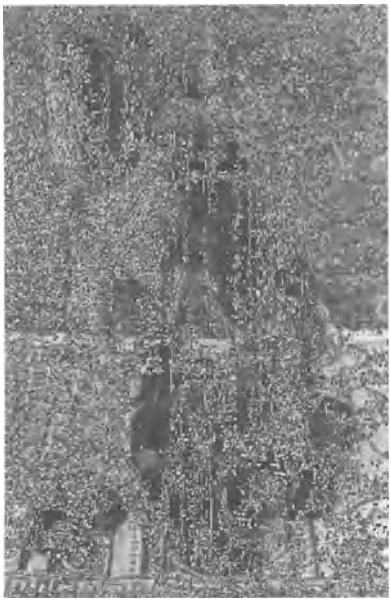

Fig. 7.3 The result of extracting the hidden picture of Figure 7.2 after JPEG compression of the cover.

in finding the transforms and adding redundancy so that the message is as undetectable as possible and as robust to further transforms of the cover (for example, compression) as possible.

Another line of research is methods to detect or defeat steganography. We have seen one method for defeating steganography, illustrated in Figure 7.3. In this case, a very simplistic steganography method was defeated by compressing the image. More sophisticated approaches are needed to detect or defeat more sophisticated steganography or watermarking technology. The book by Katzenbeisser and Petitcolas [2000] is once again a good place to learn more about these issues.

7.4 BACK DOORS

There are a large number of backdoor trojans. These are programs that open up a "back door" (also called a "trap door" by some) to the computer, allowing others to have access, bypassing the usual authentication procedures.

Loki is a program that implements an information tunnel between two machines. The idea is to use the data field in ICMP echo requests and replies to implement a login session on a remote machine. Once the Loki client is installed on a machine, a Loki server can connect to the client by sending an ICMP echo request to the machine. The machine replies with an echo reply, just as it should. However, in the packet data of the request are commands to be executed on the client, and in the reply are the results of the commands. Thus, for example, one can ask to see the password file, and this is transmitted back in the echo replies.

Loki can be detected by looking for a specific value of the sequence number (see Figure 1.5, page 16): f001 in hexadecimal, or "fool." Of course, anyone with the source code can easily change this. Another indicator is a mismatch in the number of echo replies relative to the echo requests, which is caused by relatively large data transfers in one direction. This is an example of where statistical inference can be brought to bear, to look for statistical anomalies in the numbers of requests and replies.

Loki is one example of a backdoor. Next we will look at one of the more popular and powerful ones: Back Orifice (BO).

The "legitimate" use for Back Orifice is as a remote system administration tool. Install it on your computer, and any time you need to do some administration, even when you are away from your desk, you can simply connect through Back Orifice and do whatever you need to do. Most early firewalls would pass Back Orifice packets, since the security analysts did not know to block them, and the Internet was kinder and gentler in those days. Thus, you could bypass the firewall even if your company had a policy that did not allow logins from outside.

One of the problems with Back Orifice is that it does not just give you this nice back door, but it allows anybody in. After all, the whole point is to bypass authentication. This can obviously be a serious problem.

Back Orifice was first released in 1998 by the Cult of the Dead Cow, a self-described "hacker" group. It was not the first (netbus, a similar program preceded it), but it is one of the most popular. Other similar programs are Portal of Doom, DeepThroat, Sockets de Troie (French for "Trojan Sockets"), SubSeven, Doly Trojan, RingZero, and many others. These trojans all infect various flavors of the Microsoft Windows operating system for many of the same reasons that nearly all viruses target this operating system.

Among the many things that BO and these other back doors allow one to do is to capture the mouse (and move it about at will), open and close the CD, turn on the microphone (if one is plugged in) and listen, watch every key typed, and access any files on the computer. It is quite an eye opener to watch this happening, as if by magic.

There is an interesting story involving the RingZero trojan. In late September, 1999, a number of people started noticing incoming scans to ports 80, 8080, and 3128. The pattern was curious and there was much speculation about its purpose. One thought was that it was a scan for proxies. A proxy is a gateway between networks. For example, a proxy might be used to present a single IP address to the outside, with the proxy server acting to handle the address mapping required to ensure the correct delivery of packets to the internal network.

The SANS (System Administration and Network Security) Institute sent out a call to its members to be on the lookout for these scans, and to try to find out its purpose. Finally, Ron Marcum of Vanderbilt University found a copy of the trojan on one of his machines. This is an example of the Internet community working together to detect and neutralize a threat. It is also an example of alert system administrators noticing interesting patterns in the data they were monitoring. This ability to notice interesting patterns is critical to detecting new attacks.

One method for detecting back doors on a machine is to do a port scan (for example using nmap, Section 4.9.1) and look for open ports. This assumes the

back door has opened a port. Obviously the port must be open in order to use the back door, and so equally obviously the back door program must leave the port open, right? Wrong. An interesting program is available from

http://packetstorm.securify.com/UNIX/penetration/rootkits/cd00r.c.

The program listens on the interface for a particular pattern of packets, and only upon seeing the right pattern does it open the back door. For example, the original source code watched for TCP SYN packets to the following ports (in order): 200 80 22 53 3. If packets are detected to these ports in this order, the program then opens the back door on port 5002 (all of these are configurable). The back door consists of a shell program listening on 5002, giving anyone who wishes to connect access to the computer. The program can be configured to look for these port accesses all coming from the same IP address or simply to watch for the pattern regardless of the IP addresses.

Several comments are in order here. First, since the port pattern is configurable, it is impossible to construct a signature for this kind of trojan. Further, there is no reason one couldn't modify the code to look for packets other than TCP SYN packets. For example, RESET packets are quite common and often ignored by intrusion detection systems (see Green et al. [1999] and Section 4.3.2.1). Further, why restrict oneself to TCP? How about the following as a pattern to turn on the back door?

1. A RESET packet sent to port 25 from any IP address (call it IP1).
2. An echo reply (ICMP) from IP1.
3. A UDP packet to port 53 from IP1.
4. Two echo request (ICMP) packets from the same IP address, call it IP2 (which must be different from IP1).
5. A SYN packet to port 80 from IP1 and source port above 3024.

If the preceding pattern is seen, take the sum of the source port of the first packet and the last packet and open a back door on that port number (thus allowing the attacker to specify the back door port on the fly). Since IP addresses can easily be spoofed, it would be easy to send the preceding packet sequence from a single machine. Further, if the firewall policy allows all the packets in, it is extremely unlikely that any intrusion detection system would notice the preceding pattern.

The only way to detect such a trojan on the network side is to look for unusual activity (for example, suddenly seeing connections on ports that previously had no activity). Thus, anomaly detection is essential for network security.

On the host side, it is possible to detect this trojan once it opens a port by detecting this through netstat, lsof, or a port scanner such as nmap. Also, looking at the process table can sometimes detect these trojans, but only if they are either given unfortunate names (such as "back door") or you are extremely careful to track down each and every process that your machine runs, which can be quite tedious. This particular program uses inetd to handle the network operations, and

so a wrapper program (which logs all accesses and allows only connections from specific machines to specific ports) could detect and/or defeat this program. A more sophisticated program could get around this.

7.5 MISCELLANEOUS TROJANS

There are many trojans that are simply programs that claim to do something nice (a cool screen saver, a digital birthday card, etc.) and when run actually do something very bad, like format the hard drive. These are often called viruses or worms (and sometimes they are).

McAfee and Haynes [1989] report (page 76) on a program that purported to be a graphics program but when executed erased files and taunted: "Arf arf! Got you!"

Other trojans are truly viruses or worms. One such is Happy 99. When run, it opens a window, displays fireworks and the words "Happy New Year 1999," and exits. However, whenever the computer is online and sends email, Happy 99 sends itself along, propagating itself to other computers.

Some trojans (or viruses, or worms), called "logic bombs" by some, do not do anything untoward until a prespecified time (such as Michaelangelo's birthday, Columbus day, etc.), when it suddenly "goes off."

7.5.1 Distributed Denial of Service

A distributed denial-of-service attack is a denial of service attack that is simultaneously launched by many machines against a single site. Several programs are available to implement these. I will describe four of them: Trin00, Tribe Flood Network (TFN), TFN2K, and Stacheldraht (German for Barb Wire). Most of this information is available in Criscuolo [2000].

The basic architecture of these attacks is to have a client control a set of handlers, with each handler controlling a set of agents. Each of these entities should be thought of as a machine that has been compromised and had the appropriate software installed. The client is the main attacker, but it is hidden from the victim and need not even be online at the time of the attack.

A handler is usually set up on a machine that normally has a lot of traffic, for example, a DNS server. The reason for this is to attempt to hide the traffic between the handler and its agents. As the name implies, the handler controls the agents, who perform the actual attack.

The basic attack is to flood the victim with a large number of packets, far more than the victim can handle. This has the effect of (at least) locking out legitimate users of the system and may in fact bring the victim machine down completely. The attacks are usually launched against commercial Web servers, so in addition to creating havoc, they can result in considerable lost revenue.

Although the programs described implement a certain subset of possible denial-of-service attacks, there is no reason why future programs won't expand this list.

As we will see, this kind of "upgrading" of attacks is one of the distinguishing characteristics separating these programs.

The attack consists of two phases. In the first phase, the program tries to compromise as many systems as possible. These systems will then be used in the second phase of the attack, which is the flood discussed earlier. Note that the first indication that the victim has of the attack is the attack itself. The victim has no way of detecting the first phase, since this is occurring on other machines on the Internet.

The earliest tool for setting up a distributed attack was Trin00. As with all of the tools discussed here, Trin00 affects Linux and Solaris systems. It follows the basic steps outlined earlier. In addition, it usually installs a rootkit on the compromised machines to hide the program. As a result, it is hard to detect Trin00 without specialized scanners. Look for a file called "...", which will contain a list of compromised machines. In fact, any time you find a file with this name you should be concerned. Recall that on Unix machines a filename with an initial period is "hidden", not listed without specifying the "-a" flag on ls. The directories "." (the current directory) and ".." (the previous or parent directory) are always there, and it is easy to overlook a file with one too many dots. There is no legitimate reason to have a file or directory with this name.

Trin00 attacks systems over random UDP ports, and so it is difficult to design a detector for it. It also can be configured to communicate over arbitrary ports, but it defaults to ports 27665/TCP, 27444/UDP, and 31335/UDP.

The next generation of attack tools is the Tribe Flood Network. Its handlers and agents communicate via ICMP echo reply packets. It adds a root shell on a port, allowing easy access to the system by the attacker (and anyone else).

TFN has four methods of attack:

- SYN flood (see Section 4.3.1.2).
- UDP flood. In this attack, many UDP packets are sent. The victim machine cannot handle them all, and cannot accept new connections as a result.
- Smurf (see Section 4.3.1.6).
- ICMP flood. A large number of ICMP echo requests are sent, and are too many to handle.

TFN2K adds yet another attack method, Targa3 (see Section 4.3.1.5). In addition, the attacks use spoofed addresses. In communications among the processes (client, handlers and agents), TFN2K adds decoy packets with each legitimate packet to make it difficult to backtrack to the attacker.

Finally, Stacheldraht adds to the preceding capabilities encrypted communications between client and handlers and automatic updating of agents. Stacheldraht has a limit of 6000 agents per handler.

7.6 DETECTING TROJANS

Some trojan programs are relatively easy to detect. They use a particular port to send information out or use a data field that can be checked for unusual data or particular strings. The catch is that one must know this in advance. New trojans must be detected as unusual activity (see Sections 4.5.4 and 4.8).

The problem of detection of trojans before they are activated is unexpectedly difficult. Consider the following problem: you have heard that someone has modified "login" to allow a back door. It operates exactly as it should unless the password given is a special one known only to the author of the trojan program. How can you determine that your copy of login is uncorrupted, rather than a copy of this trojan?

The first thing that may come to mind is to run "strings" (Section 5.6.1) on the login program. This is a program that scans through a binary file and prints out all segments that look as though they might be ascii text. This may work if you know what password the author of the trojan used, or if the password happens to be something suspicious such as "backdoorpassword," but is unlikely to work in real life.

But wait, you are running Linux and thus have access to the full source code! You find the source for login, pore through it, and find nothing at all suspicious! You are clean! Well, just to be sure, you recompile the program from the source. Now, you are sure you are clean and can proceed safe in the knowledge that no back door exists!

But what if the problem was in the compiler all the time, and not in the login program itself? This is illustrated by Thompson [1984], one of the originators of Unix. The basic technique is also discussed in Denning [1990b] by Witten [1990]. The idea is to place in the compiler code that compiles the trojan into the login program. Of course, this leaves the compiler with suspicious code in the source, so we compile the compiler and then change the source back. The binary for the compiler has the "bug" that will compile the back door into the login program, but the source code for both the compiler and the login program is clean. If we want to be really clever, we have the binary compiler also insert the appropriate "bug" into any newly compiled version of itself, so we cannot even save ourselves by recompiling the compiler.

Thompson points out that there is nothing magic about the compiler: the same kind of thing can be done with assemblers, linkers, and even hardware microcode (how much of your computer was manufactured in the country in which you reside?) Another approach is to modify one of the shared libraries that a program loads.

This points out one of the fundamental problems with security. If you cannot trust the people providing the systems to you, you are potentially doomed. Consider how much of the computer code written today is written overseas. Working for the U.S. Navy, I am naturally suspicious of code from foreign countries, but if you prefer, think about how much you might want to trust code written by your competitor. Further, consider how much of the code is delivered in binary format only (no source code), and hence there is no easy way to determine what it is doing or what its vulnerabilities might be.

As you can see, if one wishes to be paranoid, it is not hard at all to come up with reasons to be. Good security professionals tend to be paranoid.

7.7 FURTHER READING

There has been a lot of work in multilevel security and covert channels. Kang et al. [1997b] provide information on the design of an architecture using the pump described earlier. Kang et al. [1997a] describe a multilevel security architecture using the pump.

A very good place to start learning about steganography and digital watermarking is the book by Katzenbeisser and Petitcolas [2000]. Anderson and Petitcolas [1998] and Simmons [1998b] discuss the issues of how much information can be hidden, given the constraints of trying to make the message hard to discover and robust to degradation.

For those with a theoretical computer science background, Thimbleby et al. [1998] present a formal model for trojan programs and computer viruses. They define a trojan to be a "nonempty recursively enumerable relation $T \subseteq R \times R \times L$" that has certain formal properties. In this definition, R is the set of all "representations" (essentially the different possible states that the machine can be in or different environments in which processing is taking place) and L is a set of labels. The properties boil down to stating that a program with the same name will operate differently in two different but similar environments. Describing this work in detail would require too much of a departure into computer science background, so I leave it to the interested reader to pursue.

A discussion of malicious code and what to do about it can be found in McGraw and Morrisett [2000]. Weiss and Amoroso [1988] is an early paper describing an approach to ensuring the integrity of software written by a team. This kind of source code protection should be the minimum requirement for vendors producing code for sensitive applications, such as military, banking, or critical infrastructure.

Finally, Denning [1990b] has a number of good articles about famous attacks, malicious code, and what to do about them. We have cited several of these papers in the preceding discussion, but there are a number of others that are of interest.

Appendix A

Well-Known Port Numbers

Table A.1 Port/service pairings for some of the more common ports (1–33).

Keyword	Decimal	Description
tcpmux	1/tcp	TCP Port Service Multiplexer
tcpmux	1/udp	TCP Port Service Multiplexer
compressnet	2/tcp	Management Utility
compressnet	2/udp	Management Utility
compressnet	3/tcp	Compression Process
compressnet	3/udp	Compression Process
rje	5/tcp	Remote Job Entry
rje	5/udp	Remote Job Entry
echo	7/tcp	Echo
echo	7/udp	Echo
discard	9/tcp	Discard
discard	9/udp	Discard
systat	11/tcp	Active Users
systat	11/udp	Active Users
daytime	13/tcp	Daytime (RFC 867)
daytime	13/udp	Daytime (RFC 867)
qotd	17/tcp	Quote of the Day
qotd	17/udp	Quote of the Day
msp	18/tcp	Message Send Protocol
msp	18/udp	Message Send Protocol
chargen	19/tcp	Character Generator
chargen	19/udp	Character Generator
ftp-data	20/tcp	File Transfer [Default Data]
ftp-data	20/udp	File Transfer [Default Data]
ftp	21/tcp	File Transfer [Control]
ftp	21/udp	File Transfer [Control]
ssh	22/tcp	SSH Remote Login Protocol
ssh	22/udp	SSH Remote Login Protocol
telnet	23/tcp	Telnet
telnet	23/udp	Telnet
smtp	25/tcp	Simple Mail Transfer
smtp	25/udp	Simple Mail Transfer
dsp	33/tcp	Display Support Protocol
dsp	33/udp	Display Support Protocol

Table A.2 Port/service pairings for some of the more common ports (37–95).

Keyword	Decimal	Description
time	37/tcp	Time
time	37/udp	Time
rap	38/tcp	Route Access Protocol
rap	38/udp	Route Access Protocol
rlp	39/tcp	Resource Location Protocol
rlp	39/udp	Resource Location Protocol
graphics	41/tcp	Graphics
graphics	41/udp	Graphics
nameserver	42/tcp	Host Name Server
nameserver	42/udp	Host Name Server
nicname	43/tcp	Who Is
nicname	43/udp	Who Is
domain	53/tcp	Domain Name Server
domain	53/udp	Domain Name Server
whois++	63/tcp	whois++
whois++	63/udp	whois++
bootps	67/tcp	Bootstrap Protocol Server
bootps	67/udp	Bootstrap Protocol Server
bootpc	68/tcp	Bootstrap Protocol Client
bootpc	68/udp	Bootstrap Protocol Client
tftp	69/tcp	Trivial File Transfer
tftp	69/udp	Trivial File Transfer
gopher	70/tcp	Gopher
gopher	70/udp	Gopher
finger	79/tcp	Finger
finger	79/udp	Finger
http	80/tcp	World Wide Web HTTP
http	80/udp	World Wide Web HTTP
hosts2-ns	81/tcp	HOSTS2 Name Server
hosts2-ns	81/udp	HOSTS2 Name Server
kerberos	88/tcp	Kerberos
kerberos	88/udp	Kerberos
supdup	95/tcp	SUPDUP
supdup	95/udp	SUPDUP

Table A.3 Port/service pairings for some of the more common ports (101–139).

Keyword	Decimal	Description
hostname	101/tcp	NIC Host Name Server
hostname	101/udp	NIC Host Name Server
rtelnet	107/tcp	Remote Telnet Service
rtelnet	107/udp	Remote Telnet Service
pop2	109/tcp	Post Office Protocol - Version 2
pop2	109/udp	Post Office Protocol - Version 2
pop3	110/tcp	Post Office Protocol - Version 3
pop3	110/udp	Post Office Protocol - Version 3
sunrpc	111/tcp	SUN Remote Procedure Call
sunrpc	111/udp	SUN Remote Procedure Call
ident	113/tcp	
auth	113/tcp	Authentication Service
auth	113/udp	Authentication Service
audionews	114/tcp	Audio News Multicast
audionews	114/udp	Audio News Multicast
sftp	115/tcp	Simple File Transfer Protocol
sftp	115/udp	Simple File Transfer Protocol
nntp	119/tcp	Network News Transfer Protocol
nntp	119/udp	Network News Transfer Protocol
statsrv	133/tcp	Statistics Service
statsrv	133/udp	Statistics Service
ingres-net	134/tcp	INGRES-NET Service
ingres-net	134/udp	INGRES-NET Service
epmap	135/tcp	DCE endpoint resolution
epmap	135/udp	DCE endpoint resolution
profile	136/tcp	PROFILE Naming System
profile	136/udp	PROFILE Naming System
netbios-ns	137/tcp	NETBIOS Name Service
netbios-ns	137/udp	NETBIOS Name Service
netbios-dgm	138/tcp	NETBIOS Datagram Service
netbios-dgm	138/udp	NETBIOS Datagram Service
netbios-ssn	139/tcp	NETBIOS Session Service
netbios-ssn	139/udp	NETBIOS Session Service

Table A.4 Port/service pairings for some of the more common ports (142–565).

Keyword	Decimal	Description
imap	143/tcp	Internet Message Access Protocol
imap	143/udp	Internet Message Access Protocol
pcmail-srv	158/tcp	PCMail Server
pcmail-srv	158/udp	PCMail Server
sgmp-traps	160/tcp	SGMP-TRAPS
sgmp-traps	160/udp	SGMP-TRAPS
snmp	161/tcp	SNMP
snmp	161/udp	SNMP
snmptrap	162/tcp	SNMPTRAP
snmptrap	162/udp	SNMPTRAP
imap3	220/tcp	Interactive Mail Access Protocol v3
imap3	220/udp	Interactive Mail Access Protocol v3
yak-chat	258/tcp	Yak Winsock Personal Chat
yak-chat	258/udp	Yak Winsock Personal Chat
http-mgmt	280/tcp	http-mgmt
http-mgmt	280/udp	http-mgmt
exec	512/tcp	remote process execution;
biff	512/udp	used by mail system to notify users
login	513/tcp	remote login a la telnet;
who	513/udp	who is logged on
shell	514/tcp	cmd
syslog	514/udp	syslog
printer	515/tcp	spooler
printer	515/udp	spooler
talk	517/tcp	like tenex link, but across
talk	517/udp	like tenex link, but across
uucp	540/tcp	uucpd
uucp	540/udp	uucpd
uucp-rlogin	541/tcp	uucp-rlogin
uucp-rlogin	541/udp	uucp-rlogin
nntps	563/tcp	nntp protocol over TLS/SSL (was snntp)
nntps	563/udp	nntp protocol over TLS/SSL (was snntp)
whoami	565/tcp	whoami
whoami	565/udp	whoami

Table A.5 Port/service pairings for some of the more common ports (666–2049).

Keyword	Decimal	Description
doom	666/tcp	doom Id Software
doom	666/udp	doom Id Software
flexlm	744/tcp	Flexible License Manager
flexlm	744/udp	Flexible License Manager
kerberos-adm	749/tcp	Kerberos administration
kerberos-adm	749/udp	Kerberos administration
kerberos-iv	750/udp	Kerberos version iv
phonebook	767/tcp	phone
phonebook	767/udp	phone
accessbuilder	888/tcp	AccessBuilder
accessbuilder	888/udp	AccessBuilder
ftps-data	989/tcp	FTP protocol, data, over TLS/SSL
ftps-data	989/udp	FTP protocol, data, over TLS/SSL
ftps	990/tcp	FTP protocol, control, over TLS/SSL
ftps	990/udp	FTP protocol, control, over TLS/SSL
nas	991/tcp	Netnews Administration System
nas	991/udp	Netnews Administration System
telnets	992/tcp	telnet protocol over TLS/SSL
telnets	992/udp	telnet protocol over TLS/SSL
imaps	993/tcp	imap4 protocol over TLS/SSL
imaps	993/udp	imap4 protocol over TLS/SSL
ircs	994/tcp	irc protocol over TLS/SSL
ircs	994/udp	irc protocol over TLS/SSL
pop3s	995/tcp	pop3 protocol over TLS/SSL (was spop3)
pop3s	995/udp	pop3 protocol over TLS/SSL (was spop3)
blackjack	1025/tcp	network blackjack
blackjack	1025/udp	network blackjack
lotusnote	1352/tcp	Lotus Note
lotusnote	1352/udp	Lotus Note
shockwave	1626/tcp	Shockwave
shockwave	1626/udp	Shockwave
nfs	2049/tcp	Network File System - Sun Microsystems
nfs	2049/udp	Network File System - Sun Microsystems

Table A.6 Port/service pairings for some of the more common ports (3334–33434).

Keyword	Decimal	Description
directv-web	3334/tcp	Direct TV Webcasting
directv-web	3334/udp	Direct TV Webcasting
directv-soft	3335/tcp	Direct TV Software Updates
directv-soft	3335/udp	Direct TV Software Updates
directv-tick	3336/tcp	Direct TV Tickers
directv-tick	3336/udp	Direct TV Tickers
directv-catlg	3337/tcp	Direct TV Data Catalog
directv-catlg	3337/udp	Direct TV Data Catalog
rwhois	4321/tcp	Remote Who Is
rwhois	4321/udp	Remote Who Is
aol	5190/tcp	AmericaOnline
aol	5190/udp	AmericaOnline
aol-1	5191/tcp	AmericaOnline1
aol-1	5191/udp	AmericaOnline1
aol-2	5192/tcp	AmericaOnline2
aol-2	5192/udp	AmericaOnline2
aol-3	5193/tcp	AmericaOnline3
aol-3	5193/udp	AmericaOnline3
x11	6000-6063/tcp	X Window System
x11	6000-6063/udp	X Window System
statsci1-lm	6144/tcp	StatSci License Manager - 1
statsci1-lm	6144/udp	StatSci License Manager - 1
statsci2-lm	6145/tcp	StatSci License Manager - 2
statsci2-lm	6145/udp	StatSci License Manager - 2
http-alt	8008/tcp	HTTP Alternate
http-alt	8008/udp	HTTP Alternate
http-alt	8080/tcp	HTTP Alternate (see port 80)
http-alt	8080/udp	HTTP Alternate (see port 80)
quake	26000/tcp	quake
quake	26000/udp	quake
traceroute	33434/tcp	traceroute use
traceroute	33434/udp	traceroute use

Appendix B

Trojan Port Numbers

Table B.1 Port/trojan pairings for some of the more common trojans, ports 2–456.

Port	Trojan(s)
2	Death
21	Back Construction, Blade Runner, Doly Trojan, Fore, FTP trojan, Invisible FTP, Larva, MBT, Motiv, Net Administrator, Senna Spy FTP Server, WebEx, WinCrash
23	Tiny Telnet Server, Truva Atl
25	Aji, Antigen, Email Password Sender Gip, Happy 99, I Love You, Kuang 2, Magic Horse, Moscow Email Trojan, Naebi, NewApt, ProMail trojan, Shtrilitz, Stealth, Tapiras, Terminator WinPC, WinSpy
31	Agent 31, Hackers Paradise, Masters Paradise
41	DeepThroat
48	DRAT
50	DRAT
59	DMSetup
79	Firehotcker
80	Back End, Executor, Hooker, RingZero
99	Hidden Port
110	ProMail trojan
113	Invisible Identd Deamon, Kazimas
119	Happy 99
121	JammerKillah
123	Net Controller
133	Farnaz, Infector
146 (UDP)	Infector
170	A-trojan
421	TCP Wrappers
456	Hackers Paradise

Table B.2 Port/trojan pairings for some of the more common trojans, ports 531–1245.

Port	Trojan(s)
531	Rasmin
555	Ini-Killer, NeTAdministrator, Phase Zero, Stealth Spy
606	Secret Service
666	Attack FTP, Back Construction, NokNok, Cain & Abel, Satanz Backdoor, ServeU, Shadow Phyre
667	SniperNet
669	DP Trojan
692	GayOL
777	Aim Spy
808	WinHole
911	Dark Shadow
999	DeepThroat, WinSatan
1000	Der Spacher 3
1001	Der Spacher 3, Le Guardien, Silencer, WebEx
1010-12	Doly Trojan
1015-16	Doly Trojan
1020	Vampire
1024	NetSpy
1042	Bla
1045	Rasmin
1050	MiniCommand
1080-3	WinHole
1090	Xtreme
1095,7,8	RAT
1099	BFevolution, RAT
1170	Psyber Stream Server, Streaming Audio trojan, Voice
1200-1 (UDP)	NoBackO
1207	SoftWAR
1212	Kaos
1225	Scarab
1234	Ultors Trojan
1243	BackDoor-G, SubSeven, SubSeven Apocalypse, Tiles
1245	VooDoo Doll

Table B.3 Port/trojan pairings for some of the more common trojans, ports 1255–3024.

Port	Trojan(s)
1255	Scarab
1256	Project nEXT
1269	Mavericks Matrix
1313	NETrojan
1338	Millennium Worm
1349 (UDP)	BO DLL
1492	FTP99CMP
1509	Psyber Streaming Server
1524	Trinoo
1600	Shivka-Burka
1777	Scarab
1807	SpySender
1966	Fake FTP
1969	OpC BO
1981	Shockrave
1999	BackDoor, TransScout
2000	Der Spaeher 3, Insane Network, TransScout
2001	Der Spaeher 3, TransScout, Trojan Cow
2002-5	TransScout
2023	Ripper
2080	WinHole
2115	Bugs
2140	Deep Throat, The Invasor
2155	Illusion Mailer
2283	HVL Rat5
2300	Xplorer
2565	Striker
2583	WinCrash
2600	Digital RootBeer
2716	The Prayer
2773	SubSeven
2801	Phineas Phucker
3000	Remote Shutdown
3024	WinCrash

Table B.4 Port/trojan pairings for some of the more common trojans, ports 3128–6006.

Port	Trojan(s)
3128	RingZero
3129	Masters Paradise
3150	Deep Throat, The Invasor
3456	Terror Trojan
3459	Eclipse 2000, Sanctuary
3700	Portal of Doom
3791	Eclypse
3801 (UDP)	Eclypse
4000	Skydance
4092	WinCrash
4242	Virtual hacking Machine
4321	BoBo
4444	Prosiak, Swift remote
4567	File Nail
4590	ICQTrojan
5000	Bubbel, Back Door Setup, Sockets de Troie
5001	Back Door Setup, Sockets de Troie
5010	Solo
5011	One of the Last Trojans (OOTLT)
5031	NetMetropolitan
5321	Firehotcker
5343	wCrat
5400-2	Blade Runner, Back Construction
5550	Xtcp
5512	Illusion Mailer
5555	ServeMe
55567	BO Facil
5569	Robo-Hack
5637-8	PC Crasher
5742	WinCrash
5882 (UDP)	Y3K RAT
5888	Y3K RAT
6000	The Thing
6006	The Thing

Table B.5 Port/trojan pairings for some of the more common trojans, ports 6272–9878.

Port	Trojan(s)
6272	Secret Service
6400	The Thing
6667	Schedule Agent
6669	Host Control, Vampyre
6670	DeepThroat, BackWeb Server, WinNuke, eXtreame
6711	SubSeven
6712	Funny Trojan, SubSeven
6713	SubSeven
6723	Mstream
6771	DeepThroat
6776	2000 Cracks, BackDoor-G, SubSeven
6838 (UDP)	Mstream
6912	Shit Heep (not port 69123!)
6939	Indoctrination
6969	GateCrasher, Priority, IRC 3, NetController
6970	GateCrasher
7000	Remote Grab, Kazimas, SubSeven
7001	Freak88
7215	SubSeven
7300-1	NetMonitor
7306-8	NetMonitor
7424	Host Control
7789	Back Door Setup, ICKiller
7983	Mstream
8080	RingZero
8787	Back Orifice 2000
8897	HackOffice
8988	BacHack
8989	Rcon
9000	Netministrator
9325 (UDP)	Mstream
9400	InCommand
9872-5	Portal of Doom
9876	Cyber Attacker, RUX
9878	TransScout

Table B.6 Port/trojan pairings for some of the more common trojans, ports 9989–17300.

Port	Trojan(s)
9989	iNi-Killer
9999	The Prayer
10067 (UDP)	Portal of Doom
10085-6	Syphilis
10101	BrainSpy
10167 (UDP)	Portal of Doom
10528	Host Control
10520	Acid Shivers
10607	Coma
10666 (UDP)	Ambush
11000	Senna Spy
11050-1	Host Control
11223	Progenic trojan, Secret Agent
12076	Gjamer
12223	Hack´99 KeyLogger
12345	GabanBus, My Pics, NetBus, Pie Bill Gates, Whack Job, X-bill
12346	GabanBus, NetBus, X-bill
12349	BioNet
12361-2	Whack-a-mole
12623 (UDP)	DUN Control
12624	Buttman
12631	WhackJob
12754	Mstream
13000	Senna Spy
13010	Hacker Brazil
15092	Host Control
15104	Mstream
16660	Stacheldraht
16484	Mosucker
16772	ICQ Revenge
16969	Priority
17166	Mosaic
17300	Kuang2 The Virus

Table B.7 Port/trojan pairings for some of the more common trojans, ports 17777–31336.

Port	Trojan(s)
17777	Nephron
18753 (UDP)	Shaft
19864	ICQ Revenge
20001	Millennium
20002	AcidkoR
20034	NetBus 2 Pro, NetRex, Whack Job
20203	Chupacabra
20331	Bla
20432	Shaft
20432 (UDP)	Shaft
21544	GirlFriend, Kidterror, Schwindler, WinSp00fer
22222	Prosiak
23023	Logged
23432	Asylum
23456	Evil FTP, Ugly FTP, Whack Job
23476-7	Donald Dick
26274 (UDP)	Delta Source
26681	Spy Voice
27374	SubSeven
27444 (UDP)	Trinoo
27573	SubSeven
27665	Trinoo
29104	Host Control
29891 (UDP)	The Unexplained
30001	TerrOr32
30029	AOL Trojan
30100-3	NetSphere
30133	NetSphere
30303	Sockets de Troie
30947	Intruse
30999	Kuang2
31335 (UDP)	Trinoo
31336	Bo Whack, ButtFunnel

Table B.8 Port/trojan pairings for some of the more common trojans, ports 31337–60000.

Port	Trojan(s)
31337	Baron Night, BO client, BO2, Bo Facil
31337 (UDP)	BackFire, Back Orifice, DeepBO, Freak>
31338	NetSpy DK, ButtFunnel
31338 (UDP)	Back Orifice, DeepBO
31339	NetSpy DK
31666	BOWhack
31785,87-89,91-92	Hack´a´Tack
32100	Peanut Brittle, Project nEXT
32418	Acid Battery
33333	Blakharaz, Prosiak
33577	PsychWard
33777	PsychWard
33911	Spirit 2001a
34324	BigGluck, TN
34555 (UDP)	Trinoo (Windows)
35555 (UDP)	Trinoo (Windows)
37651	YAT
40412	The Spy
40421	Agent 40421, Masters Paradise
40422-3,6	Masters Paradise
41666	Remote Boot
44444	Prosiak
47262 (UDP)	Delta Source
50505	Sockets de Troie
50766	Fore, Schwindler
51996	Cafeini
52317	Acid Battery 2000
53001	Remote Windows Shutdown
54283	SubSeven
54320	Back Orifice 2000
54321	School Bus
54321 (UDP)	Back Orifice 2000
57341	NetRaider
58339	ButtFunnel
60000	Deep Throat

Table B.9 Port/trojan pairings for some of the more common trojans, ports 60068–65535.

Port	Trojan(s)
60068	Xzip 6000068
60411	Connection
61348	Bunker-Hill
61466	Telecommando
61603	Bunker-Hill
63485	Bunker-Hill
65000	Devil, Stacheldraht
65432	The Traitor
65432 (UDP)	The Traitor
65535	RC

Appendix C

Country Codes

Table C.1 Two character country codes, AD–FM.

Code	Country	Code	Country
AD	Andorra	CA	Canada
AE	United Arab Emirates	CC	Cocos (Keeling) Islands
AF	Afghanistan	CF	Central African Republic
AG	Antigua and Barbuda	CG	Congo
AI	Anguilla	CH	Switzerland
AL	Albania	CI	Côte D'Ivoire (Ivory Coast)
AM	Armenia	CK	Cook Islands
AN	Netherlands Antilles	CL	Chile
AO	Angola	CM	Cameroon
AQ	Antarctica	CN	China
AR	Argentina	CO	Colombia
AS	American Samoa	CR	Costa Rica
AT	Austria	CS	Czechoslovakia (former)
AU	Australia	CU	Cuba
AW	Aruba	CV	Cape Verde
AZ	Azerbaijan	CX	Christmas Island
BA	Bosnia and Herzegovina	CY	Cyprus
BB	Barbados	CZ	Czech Republic
BD	Bangladesh	DE	Germany
BE	Belgium	DJ	Djibouti
BF	Burkina Faso	DK	Denmark
BG	Bulgaria	DM	Dominica
BH	Bahrain	DO	Dominican Republic
BI	Burundi	DZ	Algeria
BJ	Benin	EC	Ecuador
BM	Bermuda	EE	Estonia
BN	Brunei Darussalam	EG	Egypt
BO	Bolivia	EH	Western Sahara
BR	Brazil	ER	Eritrea
BS	Bahamas	ES	Spain
BT	Bhutan	ET	Ethiopia
BV	Bouvet Island	FI	Finland
BW	Botswana	FJ	Fiji
BY	Belarus	FK	Falkland Islands (Malvinas)
BZ	Belize	FM	Micronesia

Table C.2 Two character country codes, FO–MN.

Code	Country	Code	Country
FO	Faroe Islands	IT	Italy
FR	France	JM	Jamaica
FX	France, Metropolitan	JO	Jordan
GA	Gabon	JP	Japan
GB	Great Britain (UK)	KE	Kenya
GD	Grenada	KG	Kyrgyzstan
GE	Georgia	KH	Cambodia
GF	French Guiana	KI	Kiribati
GH	Ghana	KM	Comoros
GI	Gibraltar	KN	Saint Kitts and Nevis
GL	Greenland	KP	Korea (North)
GM	Gambia	KR	Korea (South)
GN	Guinea	KW	Kuwait
GP	Guadeloupe	KY	Cayman Islands
GQ	Equatorial Guinea	KZ	Kazakhstan
GR	Greece	LA	Laos
GS	S. Georgia and S. Sandwich Isls.	LB	Lebanon
GT	Guatemala	LC	Saint Lucia
GU	Guam	LI	Liechtenstein
GW	Guinea-Bissau	LK	Sri Lanka
GY	Guyana	LR	Liberia
HK	Hong Kong	LS	Lesotho
HM	Heard and McDonald Islands	LT	Lithuania
HN	Honduras	LU	Luxembourg
HR	Croatia (Hrvatska)	LV	Latvia
HT	Haiti	LY	Libya
HU	Hungary	MA	Morocco
ID	Indonesia	MC	Monaco
IE	Ireland	MD	Moldova
IL	Israel	MG	Madagascar
IN	India	MH	Marshall Islands
IO	British Indian Ocean Territory	MK	Macedonia
IQ	Iraq	ML	Mali
IR	Iran	MM	Myanmar
IS	Iceland	MN	Mongolia

Table C.3 Two character country codes, MO–TJ.

Code	Country	Code	Country
MO	Macau	PR	Puerto Rico
MP	Northern Mariana Islands	PT	Portugal
MQ	Martinique	PW	Palau
MR	Mauritania	PY	Paraguay
MS	Montserrat	QA	Qatar
MT	Malta	RE	Reunion
MU	Mauritius	RO	Romania
MV	Maldives	RU	Russian Federation
MW	Malawi	RW	Rwanda
MX	Mexico	SA	Saudi Arabia
MY	Malaysia	Sb	Solomon Islands
MZ	Mozambique	SC	Seychelles
NA	Namibia	SD	Sudan
NC	New Caledonia	SE	Sweden
NE	Niger	SG	Singapore
NF	Norfolk Island	SH	St. Helena
NG	Nigeria	SI	Slovenia
NI	Nicaragua	SJ	Svalbard and Jan Mayen Isls.
NL	Netherlands	SK	Slovak Republic
NO	Norway	SL	Sierra Leone
NP	Nepal	SM	San Marino
NR	Nauru	SN	Senegal
NT	Neutral Zone	SO	Somalia
NU	Niue	SR	Suriname
NZ	New Zealand (Aotearoa)	ST	Sao Tome and Principe
OM	Oman	SU	USSR (former)
PA	Panama	SV	El Salvador
PE	Peru	SY	Syria
PF	French Polynesia	SZ	Swaziland
PG	Papua New Guinea	TC	Turks and Caicos Islands
PH	Philippines	TD	Chad
PK	Pakistan	TF	French Southern Territories
PL	Poland	TG	Togo
PM	St. Pierre and Miquelon	TH	Thailand
PN	Pitcairn	TJ	Tajikistan

Table C.4 Two character country codes, TK–ZW.

Code	Country
TK	Tokelau
TM	Turkmenistan
TN	Tunisia
TO	Tonga
TP	East Timor
TR	Turkey
TT	Trinidad and Tobago
TV	Tuvalu
TW	Taiwan
TZ	Tanzania
UA	Ukraine
UG	Uganda
UK	United Kingdom
UM	US Minor Outlying Islands
US	United States
UY	Uruguay
UZ	Uzbekistan
VA	Vatican City State (Holy See)
VC	Saint Vincent and the Grenadines
VE	Venezuela
VG	Virgin Islands (British)
VI	Virgin Islands (U.S.)
VN	Viet Nam
VU	Vanuatu
WF	Wallis and Futuna Islands
WS	Samoa
YE	Yemen
YT	Mayotte
YU	Yugoslavia
ZA	South Africa
ZM	Zambia
ZR	Zaire
ZW	Zimbabwe

Appendix D

Security Web Sites

D.1 INTRODUCTION

In this section, I list a number of Web sites that are of interest for computer security and intrusion detection purposes. This listing is not complete, but is a good starting point. At the time of this writing (January-April 2001) all of these Web sites were active, but I cannot guarantee they will be in the future.

I do not endorse any of these pages. This is a list of pages that I have found interesting or useful or that I have simply come across while Web surfing. By the same token, I do not claim that it is in any way complete. There are bound to be important sites I have left off. A Web search will no doubt turn up many sites of interest that are not on the list.

A note of caution is in order here. Most of these Web sites are legitimate and any software provide by them is probably safe (but I do not make any guarantees here). Care should always be taken when obtaining software from the Web, particularly executables. In fact, I would not recommend using any software obtained from the Web that is not provided as source code. Further, some of these Web pages are self-described "hacker" sites. A few of these may interpret your access of their site as an invitation for them to visit your machine.

I have organized these sites into rough categories. There will be some duplication because some sites fit in more than one category. Within categories there is no ordering. This may be somewhat annoying to the reader but the listings here are short enough that it should cause no major problems. Note that in typing these Web addresses in, case can be important, particularly for directories. Also beware of "zeros" that look like "Os."

First, there is a list of general information Web sites (including sites that do not fit well into the other categories) and then a listing of security-related Web sites. A listing of Web sites providing information about "cyber crime" is also provided as well as sites where the software described in the book is available and where data may be obtained. Finally is a list of Web sites that are specifically aimed at intrusion detection.

D.2 GENERAL

Miscellaneous Web sites with useful information relevant to computer security, statistical analysis, and data visualization follow.

freshmeat.net A repository of software for Linux.

slashdot.org "News for Nerds," with a focus on Linux.

www.nd.edu/~networks/visual/table.html A collection of examples of network visualization.

www.caida.org/tools/ Tools and software for visualization.

www.isoc.org/ Internet Society.

www.ietf.org/ Internet Engineering Task Force.

lib.stat.cmu.edu/R/CRAN/contents.html The R repository at Carnegy Mellon, where the R distribution, documentation, and contributed packages are available.

www.R-project.org The R Project. The official Web site for R.

lark.cc.ukans.edu/~pauljohn/R/statsRus.html A tip sheet for the R language.

www.bell-labs.com/topic/societies/asagraphics/resources.html A collection of Online resources for statistical graphics.

hotspur.psych.yorku.ca/SCS/Gallery/intro.html A gallary of data visualization examples, both good and bad.

aleph0.clarku.edu/~djoyce/java/Phyltree/cover.html A discussion of phylogeny and reconstructing phylogenetic trees. Includes a Java applet.

www.cs.bell-labs.com/who/ches/map/ Bill Cheswick's Internet Mapping Project.

www.mids.org/weather/ The Internet Weather Report.

www.cybergeography.org Cybergeography research and information.

www.mappingcyberspace.com Mapping cyberspace book and information.

www.viruslist.com Information about viruses.

www.isc.org The Internet Software Consortium.

www.slac.stanford.edu/grp/scs/net/talk/escc-sdo-apr97/meas/ppframe.htm Internet End-to-end monitoring and performance measuring.

www.internettrafficreport.com The Internet Traffic Report. Monitoring the flow of data around the world.

www.netsizer.com Evaluating the size of the Internet.

www.fnc.gov The Federal Networking Council.

www.fnc.gov/claffy.html Internet measurement tools.

www.sims.berkeley.edu/resources/infoecon/Accounting.html Accounting and measurement of Internet traffic.

www.mit.edu/people/mkgray/net/ Statistics on the growth of the Internet.

www.science.uva.nl/~mes/jargon/ A jargon dictionary useful for tracking down the definitions of unfamiliar jargon.

www.uni-paderborn.de/cs/ag-klbue/staff/murray/work/publications/icc01.p-df An interesting paper entitled "Visualization of Traffic Structures," by Oliver Niggemann, Benno Stein and Jens Tölle. The "-" after the "p" is not part of the URL.

members.aol.com/edswing/flodar/flodarviz.html A description of FLODAR, a visualization tool for network traffic, developed at the National Security Agency (according to the Web site). There are a few references listed here that might be of interest.

dmoz.org The Open Directory Project. The goal of this project is to have the most comprehensive directory of the Web. Most relevant to this book is the subdirectory

at **dmoz.org/Computers/**.

jeff.cs.mcgill.ca/~godfried/teaching/pr-web.html Pattern recognition resources and information on the Web.

www.mpi-sb.mpg.de/~mutzel/alcom-it/alcomgdraw.html Graph drawing tools.

www.ics.uci.edu/~eppstein/gina/gdraw.html More graph drawing tools, links and information.

rw4.cs.uni-sb.de/users/sander/html/gstools.html More graph drawing tools and links.

www.contrib.andrew.cmu.edu/~krack/ KrackPlot: a social network visualization program. Might be of interest for visualizing networks and attacks.

www.mpi-fg-koeln.mpg.de/~lk/netvis.html Another site devoted to (social) network visualization.

D.3 SECURITY

Web sites with a security focus follow.

www.cert.org CERT coordination center.

www.sans.org System Administration and Network Security.

xforce.iss.net Internet Security Systems page of vulnerabilities and information.

www.fish.com/security/ Some of Dan Farmer's security-related papers.

www.practicalsecurity.com Computer security information. Contains a list of links.

www.insecure.org Computer security information and the Nmap scanner.

www.packetfactory.net Network security clearinghouse. A number of interesting papers are available here.

packetstorm.securify.com Information security database. Many of the main computer security tools are available here.

seclab.cs.ucdavis.edu Computer Security Research Laboratory at UC Davis.

csrc.ncsl.nist.gov Computer security resource clearinghouse for the National Institute of Standards and Technology.

csrc.ncsl.nist.gov/tools/tools.htm Unix host and network security tools.

www.infowar.com Information Warfare homepage.

www.2600.com Homepage of the 2600 magazine, the "hacker quarterly."

www.itpolicy.gsa.gov The U.S. General Services Administration IT page.

www.cit.nih.gov/security.html National Institutes of Health Internet security page.

www.nswc.navy.mil/ISSEC NSWC information security site.

ee.lbl.gov Lawrence Berkeley National Laboratory Network Research Group homepage.

www.issa-intl.org International Information Systems Security Association.

www.usenix.org Usenix, the Advanced Computing Systems Association.

www.gocsi.com Computer Security Institute.

cve.mitre.org Mitre's common vulnerabilities and exposures page.

www.rootshell.com A repository of software to exploit vulnerabilities, which contains information about vulnerabilities and news related to computer security.

www.iss.net Internet Security Systems Web site.

www.mountainwave.com Computer security news site.

www.ntbugtraq.com NT Bugtraq.

www.nsi.org/compsec.html Security resource net's computer security site. Includes a security glossary and several FAQs and papers.

www.boran.com/security/ IT security cookbook.

www.securityfocus.com News and information about computer security. Contains "bugtraq," a listing of the current bugs and vulnerabilities for various operating systems.

www.l0pht.com Information, software, and vulnerability reports. To quote their banner: "That vulnerability is completely theoretical" - Microsoft. L0pht, making

the theoretical practical since 1992. (Note: the "0" in the URL is a zero.)

www.first.org Forum of Incident Response and Security Teams.

www.nipc.gov National Infrastructure Protection Center.

www.cs.purdue.edu Purdue Computer Science Department.

www.cs.purdue.edu/coast/coast.html Computer Operations, Audit, and Security Technology.

www.cerias.purdue.edu Center for Education and Research in Informaiont Assurance and Security, Purdue University.

www.isse.gmu.edu/~csis/ Center for Secure Information Systems at George Mason University.

www.issl.org Information Systems Security Laboratory at Iowa State University.

www.infosec.jmu.edu/ Information Security Program at James Madison University.

www.cs.ucsb.edu Department of Computer Science at the University of California, Santa Barbara.

www.cl.cam.ac.uk The Computer Laboratory of the University of Cambridge.

www.cs.uidaho.edu Computer Science Department of the University of Idaho.

www.cs.uow.edu.au School of Information Technology and Computer Science, University of Wollongong, Australia.

www.cs.umbc.edu Computer Science Department, University of Maryland Baltimore County.

www.niss.org National Institute of Statistical Sciences.

www.happyhacker.org/ The Happy Hacker.

www.trusecure.net/html/tspub/hypeorhot/index.shtml TruSecure's so called "Hype or Hot" site. Lots of information about viruses and worms.

www.virusbtn.com/ The Virus Bulletin. Lots of information on viruses.

www.mitre.org/pubs/edge/february_01/ Mitre's "The Edge" Newsletter issue on Information Assurance.

www.cert.org/kb/aircert/ AirCERT. A project involving the placement of sensors on various networks attached to the Internet.

attrition.org Information about computer security and vulnerabilities.

doe-is.llnl.gov/ConferenceProceedings/DOECompSec97/DOEConf97.html The proceedings of the 1997 DOE Information Security Conference.

dmoz.org/Computers/Security/ The Open directory listing for information relevant to computer security.

project.honeynet.org/ The Web page for the HoneyNet Project. Contains a number of "Know Your Enemy" papers, and information about "blackhats" (attackers).

www.simovits.com A consulting firm that has information related to security and intrusion detection (see the link to their article archive). In particular, there is a list of the ports used by trojan horses.

www.wias.net Windermere's information assurance page.

www.Vmyths.com A site devoted to virus myths and hoaxes.

www.whitehats.com A resource for security information and recent security related news.

D.4 CRIME

These Web sites focuse on the legal and law enforcement aspects of computer security.

www.htcia.org High Technology Crime Investigation Association.

www.gahtan.com/cyberlaw/ The Cyberlaw Encyclopedia. Information about law as it applies to computers, networks and the Internet.

www.forensics.com Computer Forensics Inc. Web page containing a number of interesting case studies and documents on computer forensics.

www.computer-forensics.com Company Web page with some articles and news stories.

www.usdoj.gov/criminal/cybercrime/index.html Cybercrime page for the U.S. Department of Justice.

D.5 SOFTWARE

The following sites contain software and publications available on the Web.

www.cs.tut.fi/~rammer/aide.html Home page for the aide software.

lcamtuf.hack.pl Home of p0f.

rs.internic.net/whois.html Internic whois server.

www.nsiregistry.com/whois/ VeriSign whois server.

www.iana.org/cctld/cctld-whois.htm A page providing the Internet Assigned Numbers Authority whois information.

www.betterwhois.com/ A whois server.

www.networksolutions.com/cgi-bin/whois/whois The whois server run by Network Solutions.

www.whois.net/ Another whois server.

www.ssh.fi/ Home of secure shell.

www.snort.org Home of snort.

www.cs.umbc.edu/cadip/pubs.html University of Maryland Baltimore County, Center for Architectures for Data-Driven Information Processing publications.

www.cs.unm.edu/~immsec/papers.htm University of New Mexico papers on computer immunology.

www.niss.org/downloadabletechreports.html National Institute of Statistical Science technical reports.

www.enteract.com/~lspitz/papers.html White papers and publications on security.

www.switch.ch/docs/ttl_default.html Paper on default TTL values.

www.firstmonday.dk/issues/issue2_5/rowland/ A paper discussing covert channels in TCP/IP.

setiathome.ssl.berkeley.edu SETI at home Web page.

www.research.ibm.com/antivirus/SciPapers.htm Anitvirus Research papers, at IBM.

D.6 DATA

This section lists data available on the Web. There is not much data available, primarily because of the large volume of such data but also partly a result of privacy concerns. However, some organizations have made some of their data available.

www.schonlau.net/ Matt Schonlau's "masquerading user" data.

www.ics.uci.edu/~mlearn/MLRepository.html UCI machine learning repository. A lot of data sets used by the machine learning community reside here. One or two of these are of interest for the computer security/intrusion detection community.

kdd.ics.uci.edu/ Knowledge Discovery Database. Some data sets of Web accesses.

kdd.ics.uci.edu/databases/kddcup99/kddcup99.html 1999 data mining competition data. Task: build a system to detect intrusions in the data provided.

www.cs.unm.edu/~immsec/research.htm The Computer Immune System. System call data sets.

www.ll.mit.edu/IST/ideval/ Information on the DARPA intrusion detection systems evaluation, including points of contact for access to data.

www.virusbtn.com/Prevalence/ Index of virus bulletin prevalence tables. Historical and current information on the prevalance of computer viruses.

D.7 INTRUSION DETECTION

The following Web sites have an intrusion detection focus. Many of the sites listed above in the security section (Section D.3) also have pages devoted to intrusion detection.

packetstorm.securify.com Papers, code, and articles for and about computer security.

www.snort.org Home page for snort.

seclab.cs.ucdavis.edu The Computer Security Research Laboratory at UC Davis.

olympus.cs.ucdavis.edu/cidf/ The DARPA Common Intrusion Detection Framework.

www.isi.edu/gost/cidf/ Common Intrusion Detection Framework.

www-rnks.informatik.tu-cottbus.de/~sobirey/ids.html Michael Sobirey's intrusion detection systems page. A list of links to IDS systems.

www.robertgraham.com/pubs/network-intrusion-detection.html Intrusion detection FAQ.

www.research.ibm.com/journal/sj/371/boulanger.html "Catapults and grappling hooks: The tools and techniques of information warfare," by A. Boulanger.

www.cert.org/tech_tips/ Technical tips on Internet security issues.

www.cert.org/tech_tips/intruder_detection_checklist A good step-by-step checklist for determining whether a system has been compromised.

ftp://research.att.com/dist/internet_security/berferd.ps The paper "An evening with Berferd: In which a cracker is lured, endured, and studied," by Bill Cheswick.

www.forensics.com Computer Forensics Inc. Web page.

www.computer-forensics.com Company Web site.

www.sdl.sri.com/intrusion/index.html SRI's intrusion detection page.

www.sdl.sri.com/emerald SRI's EMERALD information site.

www.cerias.purdue.edu/coast/intrusion-detection/ Purdue's intrusion detection page.

www.nswc.navy.mil/ISSEC/CID/ NSWC intrusion detection and network security page, home of SHADOW.

www.dshield.org Distributed IDS. A free service providing a platform for sharing intrusion information.

www.sei.cmu.edu/publications/documents/99.reports/99tr028/99tr028abstract.html A technical report entitled "State of the Practice of Intrusion Detection Technologies." The "-" in the word "abstract" is not part of the URL.

Bibliography

P. Abry and D. Veitch. Wavelet analysis of long-range-dependent traffic. *IEEE Transactions on Information Theory*, 44:2–15, 1998.

R. G. Addie, M. Zukerman, and T. Neame. Fractal traffic: Measurements, modelling and performance evaluation. In *INFOCOM '95. Fourteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 977–984, 1995.

H. Akaike. A new look at statistical model identification. *IEEE Transactions on Automatic Control*, 19:716–723, 1974.

A. R. Ali. Software patching in the spc environment and its impact on switching system reliability. *IEEE Journal on Selected Areas in Communications*, 9:626–631, 1991.

E. Amoroso. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back, Traps, and Response*. Intrusion.net Books, Sparta, New Jersey, 1999.

D. Anderson, T. F. Lunt, H. Javitz, A. Tamaru, and A. Valdes. Detecting unusual program behavior using the statistical component of the next-generation intrusion detection expert system (nides). Technical Report SRI-CSL-95-06, SRI International, May, 1995.

E. Anderson. The irises of the gaspe peninsula. *Bulletin of the American Iris Society*, 59:2–5, 1935.

R. Anderson and F. A. P. Petitcolas. On the limits of steganography. *IEEE Journal on Selected Areas in Communications*, 16(4):474–481, 1998.

H. Andersson. Limit theorems for a random graph epidemic model. *The Annals of Applied Probability*, 9(4):1331–1349, 1998.

H. Andersson and T. Britton. Heterogeneity in epidemic models and its effect on the spread of infection. *Journal of Applied Probability*, 35:651–661, 1998.

H. Andersson and T. Britton. *Stochastic Epidemic Models and Their Statistical Analysis*. Springer, New York, 2000.

J. Andrews, Paul P. and M. B. Peterson, editors. *Criminal Intelligence Analysis*. Palmer Enterprises, Loomis, California, 1990.

anonymous. *Maximum Security*. Sams.net Publishing, Indianapolis, IN, 1997.

I. Ashmanov and N. Kasperskaya. The virus encyclopedia: Reaching a new level of information comfort. *IEEE Multimedia*, 6(3):81–84, July–Sept, 1999.

S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the 6th ACM Conference on Computer and Communciations Security*, pages 1–7, 1999.

S. Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *Transactions on Information Systems Security*, pages 186–205, 2000.

R. G. Bace. *Intrusion Detection*. MacMillan Technical Publishing, Indianapolis, IN, 2000.

J. S. Balasubramaniyan and J. O. Garcia-Fernandez. An architicture for intrusion detection using autonomous agents. In *Proceedings of the* 14^{th} *Annual Computer Security and Applications Conference*, pages 13–24, 1998.

T. Bass. Intrusion detection systems and multisensor data fusion. *Communications of the ACM*, 43:99–105, 2000.

D. S. Bauer, F. R. Eichelman, II, R. M. Herrera, and A. E. Irgon. Intrusion detection: An application of expert systems to computer security. In *Proceedings of the 1989 International Carnahan Conference on Security Technology*, pages 97–100, 1989.

J. Bertin. *Sémiologie Graphique*. Editions Gauthier-Villars, Paris, 1967. (English translation by W. J. Berg as *Semiology of Graphics*, University of Wisconsin Press, Madison, 1983).

P. J. Bickel and J. A. Yahav. On estimating the total probability of the unobserved outcomes of an experiment. In J. van Ryzin, editor, *Adaptive Statistical Procedures and Related Topics*, pages 332–337. Institute of Mathematical Statistics, Hayward, CA, 1986.

M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9:131–152, 1996.

S. Bleha and D. Gillespie. Computer user identification using the mean and the median as features. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 4379–4381, 1998.

S. Bleha, C. Slivinsky, and B. Hussien. Computer-access security systems using keystroke dynamics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(12):1217–1222, 1990.

S. A. Bleha and M. S. Obaidat. Dimensionality reduction and feature extraction applications in identifying computer users. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(2):452–456, 1991.

R. Brackney. Cyber-intrusion response. In *Proceedings of the Seventeenth IEEE Symposium on Reliable Distributed Systems*, pages 413–415, 1998.

K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson. Detecting disruptive routers: A distributed network monitoring approach. *IEEE Network*, 12(5):50–60, 1998.

L. Brieman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman & Hall/CRC, Boca Raton, 1998.

M. Brown and S. J. Rogers. A practical approach to user authentication. In *IEE Colloquium on Image Processing for Biometric Measurement*, pages 5/1–5/6, 1994.

R. Browne. Mode security: An infrastructure for covert channel suppression. In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privace*, pages 39–55, 1994.

R. Büschkes, D. Kesdogan, and P. Reichl. How to increase security in mobile networks by anomaly detection. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 3–12, 1998.

J. B. D. Cabrera, B. Ravichandran, and R. K. Mehra. Statistical traffic modeling for network intrusion detection. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunicaton Systems*, pages 466–473, 2000.

J. Cao, D. Davis, S. Vander Wiel, and B. Yu. Time-varying network tomography: Router link data. *Journal of the American Statistical Association*, 95(452): 1063–1075, 2000.

E. Casey. *Digital Evidence and Computer Crime*. Academic Press, San Diego, CA, 2000.

A. Chao. On estimating the probability of discovering a new species. *The Annals of Statistics*, 9(6):1339–1342, 1981.

B. Cheswick. An evening with Berferd, in which a cracker is lured, endured, and studied. In *Proceedings of the Winter USENIX Conference*, 1992.

S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford-Chen, R. Yip, and D. Zerkle. The design of GrIDS: A graph-based intrusion detection system. Technical report, Department of Computer Science, University of California at Davis, 1999.

C. Y. Chung, M. Gertz, and K. Levitt. Demids: A misuse detection system for database security. *Integrity and Internal Control in Information Systems*, 1999. Available at www.cs.umbc.edu/cadip/pubs.html.

F. Cohen. Computer viruses, theory and experiments. *Computers and Security*, 6: 22–35, 1987.

F. Cohen. Current best practice against computer viruses. In *IEEE International Carnahan Conference on Security Technology*, pages 261–270, 1991.

W. Cohen. Fast effective rule induction. In *Machine Learning: The 12th International Conference*. Morgan Kaufmann, San Francisco, CA, 1995.

D. E. Comer. *Internetworking with TCP/IP*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

L. J. Cowen and C. E. Priebe. Approximate distance clustering. *Computing Science and Statistics*, 29:337–346, 1997a.

L. J. Cowen and C. E. Priebe. Randomized nonlinear projections uncover high dimensional structure. *Advances in Applied Mathematics*, 9:319–331, 1997b.

P. J. Criscuolo. Distributed denial of service: Trin00, tribe flood network, tribe flood network 2000, and stacheldraht. Technical Report UCRL-ID-136939, Computer Incident Advisory Capability, U.S. Department of Energy, 2000. Available at ciac.llnl.gov.

M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5:835–846, 1997.

D. J. Daley and J. Gani. *Epidemic Modelling: An Introduction*. Cambridge University Press, Cambridge, UK, 2000.

B. Das and V. Bharghavan. Routing in ad-hoc networks using minimum connected dominating sets. In *Proceedings of the Sixth Annual IEEE International Conference on Computer Communications*, pages 376–380, 1997.

R. Davis. Exploring computer viruses. In *Fourth Aerospace Computer Security Applications Conference*, pages 7–11, 1988.

A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm (with discussion). *Journal of the Royal Statistical Society, B*, 39:1–38, 1977.

D. E. Denning. *Information Warfare and Security*. Addison-Wesley, Reading, MA, 1999.

D. E. Denning and P. J. Denning. *Internet Besieged: Countering Cyberspace Scofflaws*. Addison-Wesley, Reading, MA, 1998.

P. J. Denning. Computer viruses. In P. J. Denning, editor, *Computers Under Attack: Intruders, Worms, and Viruses*, pages 285–292. Addison Wesley, New York, 1990a.

P. J. Denning, editor. *Computers Under Attack: Intruders, Worms, and Viruses*. Addison-Wesley, New York, 1990b.

P. J. Denning. The Internet worm. In P. J. Denning, editor, *Computers Under Attack: Intruders, Worms, and Viruses*, pages 193–200. Addison-Wesley, New York, 1990c.

L. Devroye, L. Györfi, and G. Lugosi. *A Probabilistic Theory of Pattern Recognition*. Springer, New York, 1996.

P. D'haeseleer, S. Forrest, and P. Helman. A distributed approach to anomaly detection. Available at www.cs.unm.edu/~immsec/papers.htm, 1997.

G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, editors. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.

I. H. Dinwoodie. Conditional expectations in network traffic estimation. *Statistics & Probability Letters*, 47(1):99–103, 2000.

M. Dodge and R. Kitchin. *Mapping Cyberspace*. Routledge, London, 2001.

R. Doverspike and I. Saniee, editors. *Journal of Heuristics, Special issue: Heuristic Approaches for Telecommunciations Network Management, Planning and Expansion*, volume 6. Kluwer Academic Publishers, Amsterdam,, April, 2000.

R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley & Sons, New York, 2000.

W. DuMouchel. Computer intrusion detection based on bayes factors for comparing command transition probabilities. Technical Report 91, National Institute of Statistical Sciences, 1999. Available at www.niss.org/downloadabletechreports.html.

W. DuMouchel and M. Schonlau. A fast computer intrusion detection algorithm based on hypothesis testing of command transition probabilities. In *Proceedings of the Fourth International Conference of Knowledge Discovery and Data Mining*, pages 189–193, 1998. Available at www.niss.org/downloadabletechreports.html.

R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, Cambridge, UK, 1999.

R. Durst, T. Champion, B. Witten, E. Miller, and L. Spagnuolo. Testing and evaluating computer intrusion detection systems. *Communications of the ACM*, 42(7):53–61, 1999.

M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences*, 95:14863–14868, 1998.

T. Eisenberg, D. Gries, J. Hartmanis, D. Holcomb, M. S. Lynn, and T. Santoro. The Cornell commission: On Morris and the worm. *Communications of the ACM*, 32:706–709, 1989.

S. Elbaum and J. C. Munson. Intrusion detection through dynamic software measurement. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 41–50, 1999.

D. Endler. Intrusion detection. Applying machine learning to solaris audit data. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 268–279, 1998.

T. Escamilla. *Intrusion Detection: Network Security Beyond the Firewall.* John Wiley & Sons, Inc., New York, 1998.

D. F. Ettema and H. J. P. Timmermans, editors. *Activity-Based Approaches to Travel Analysis.* Elsevier Science Ltd., Oxford, UK, 1997.

B. S. Everitt. *Cluster Analysis.* John Wiley & Sons, Inc., New York, third edition, 1993.

D. Farmer and W. Venema. Improving the security of your site by breaking into it, 1993. Available at www.trouble.org/security.

V. Fedorov and D. Flanagan. Optimal monitoring of computer networks. In N. Flournoy, W. F. Rosenberger, and W. K. Wong, editors, *New Developments and Applications in Experimental Design*, volume 34 of *Lecture Notes-Monograph Series*, pages 1–10. Institute of Mathematical Statistics, 1998. Selected Proceedings of a 1997 Joint AMS-IMS-SIAM Summer Conference.

A. Feldmann, A. C. Gilbert, P. Huang, and W. Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *Proceedings of the ACM/SIGCOMM'99*, 1999.

A. Feldmann, A. C. Gilbert, W. Willinger, and T. G. Kurtz. Looking behind and beyond self-similarity: On scaling phenomena in measured WAN traffic. In *Proceedings of the 35th Annual Allerton Conference on Communication, Control and Computing*, pages 269–280, 1997.

A. Feldmann, A. C. Gilbert, W. Willinger, and T. G. Kurtz. The changing nature of network traffic: Scaling phenomena. *ACM SIGCOMM Computer Communications Review*, 28:5–29, 1998.

S. J. Finch, N. R. Mendell, and H. Thode, JR. Probabilistic measures of adequacy of a numerical search for a global maximum. *Journal of the American Statistical Association*, 84(408):1020–1023, 1989.

P. M. Fiorini. On modeling concurrent heavy-tailed network traffic sources and its impact on QoS. In *1999 IEEE Conference on Communications*, pages 716–720, 1999.

E. A. Fisch, G. B. White, and U. W. Pooch. The design of an audit trail analysis tool. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 126–132, 1994.

R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(Part II):179–188, 1936.

S. Floyd and V. Paxson. Why we don't know how to simulate the Internet, 1999. Available at www.aciri.org.

S. Forrest and S. A. Hofmeyr. Immunology as information processing. In L. A. Segel and I. Cohen, editors, *Design Prinicples for the Immune System and Other Distributed Autonomous Systems*, Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press, Oxford, UK, In press. Also available at www.cs.unm.edu/~forrest/ism_papers.htm.

S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40:88–96, 1997.

S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *1996 IEEE Symposium on Computer Security and Privacy*, 1996. Also available at www.cs.unm.edu/~forrest/isa_papers.htm.

S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *1994 IEEE Symposium on Research in Security and Privacy*, 1994. Also available at www.cs.unm.edu/~forrest/isa_papers.htm.

L. R. Foulds. *Graph Theory Applications*. Springer-Verlag, New York, 1992.

D. H. Freedman and C. C. Mann. *@ Large: The Strange Case of the World's Biggest Internet Invasion*. Simon & Schuster, New York, 1997.

K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, San Diego, second edition, 1990.

Fyodor. Remote OS detection via TCP/IP stack fingerprinting, 1999. Available at www.insecure.org/nmap/nmap-fingerprinting-article.html.

L. Garber. Melissa virus creates a new type of threat. *Computer*, 32(6):16–19, 1999.

A. K. Ghosh, A. Schartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 51–62, 1999.

A. C. Gilbert, W. Willinger, and A. Feldmann. Visualizing multifractal scaling behavior: A simple coloring heuristic. In *Proceedings of the 32nd Asilomar Conference on Signals, Systems and Computers*, 1998.

M. Gilfix. An integrated software immune system: A framework for automated network management, system health, and security. In *Conference on Local Computer Networks, LCN '99*, pages 254–255, 1999.

L. Girardin. An eye on network intruder-administrator shootouts. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 19–28, 1999.

L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin. Constructing computer virus phylogenies. In D. Hirschberg and G. Myers, editors, *Combinatorial Pattern Matching*, pages 253–270. Springer, New York, 1991.

L. A. Goldberg, P. W. Goldberg, C. A. Phillips, and G. B. Sorkin. Constructing computer virus phylogenies. *Journal of Algorithms*, 26:188–208, 1998.

E. Goldstein, editor. *2600: The Hacker Quarterly*, volume 17. 2600 Enterprises, Inc., 2000.

J. Green, D. Marchette, S. Northcutt, and B. Ralph. Analysis techniques for detecting coordinated attacks and probes. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 1–9, 1999.

K. Hafner and J. Markoff. *Cyberpunk: Outlaws and Hackers on the Computer Frontier*. Simon & Schuster, New York, 1995.

E. A. Hall. *Internet Core Protocols: The Definitive Guide*. O'Reilly & Associates, Sebastopol, CA, 2000.

D. J. Hand. *Construction and Assessment of Classification Rules*. John Wiley & Sons, New York, 1997.

T. W. Haynes, S. T. Hedetniemi, and P. J. Slater, editors. *Domination in Graphs: Advanced Topics*. Marcel Dekker, Inc., New York, 1998.

S. Hedberg. Combating computer viruses: IBM's new computer immune system. *IEEE Parallel & Distributed Technology: Systems & Applications*, 4(2):9–11, Summer, 1996.

P. Helman and G. Liepins. Statistical foundations of audit trail analysis for the detection of computer misuse. *IEEE Transactions on Software Engineering*, 19 (9):886–901, 1993.

Herodotus. *The Histories*. Oxford University Press, Oxford, UK, Oxford, 1998. Translated by Robin Waterfield.

C. Herringshaw. Detecting attacks on networks. *Computer*, 30(12):16–17, December, 1997.

H. W. Hethcote. The mathematics of infectious diseases. *Siam Review*, 42(4): 599–653, 2000.

H. J. Highland. Random bits & bytes. *Computers & Security*, 7:337–346, 1988.

H. J. Highland. Random bits & bytes. *Computers & Security*, 7:460–481, 1989.

H. J. Highland. The brain virus: Fact and fantasy. In P. J. Denning, editor, *Computers Under Attack: Intruders, Worms, and Viruses*, pages 293–298. Addison-Wesley, Reading, MA, 1990a.

H. J. Highland. Computer viruses - a postmortem. In P. J. Denning, editor, *Computers Under Attack: Intruders, Worms, and Viruses*, pages 299–315. Addison-Wesley, Reading, MA, 1990b.

S. A. Hofmeyr and S. Forrest. Immunology by design: An artificial immune system. In *Proceedings of the Genetics and Evolutionary Computation Conference*, pages 1289–1296, 1999.

S. A. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation Journal*, 2000. In press. Also available at www.cs.unm.edu/~forrest/isa_papers.htm.

S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

R. V. Hogg and A. T. Craig. *Introduction to Mathematical Statistics*. Prentice-Hall, Englewood Cliffs, NJ, fifth edition, 1995.

J. Hruska. Virus detection. In *European Conference on Security and Detection*, pages 128–130, 1997.

J. R. Hughes. Conservation of flow as a security mechanism in network protocols. Master's thesis, Purdue University, June, 2000.

J. R. Hughes, T. Aura, and M. Bishop. Using conservation of flow as a security mechanism in network protocols. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 132–141, 2000.

D. J. Icove. Collaring the cybercrook: An investigator's view. *IEEE Spectrum*, 34 (6):31–36, June, 1997.

R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *JCGS*, 5(3):299–314, 1996.

K. Ilgun. USTAT: A real-time intrusion detection system for Unix. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 16–28, 1993.

A. Inselberg. The plane with parallel coordinates. *The Visual Computer*, 1:69–91, 1984.

T. Jamil. Steganography: the art of hiding information in plain sight. *IEEE Potentials*, 18:10–12, 1999.

H. Javitz and A. Valdes. The SRI IDES statistical anomaly detector. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 316–326, 1991.

H. Javitz and A. Valdes. The NIDES statistical component: description and justification. Technical report, SRI International, 1993. available at www.sdl.sri.com/nides/index5.html.

C. L. Jeffrey, editor. *Program Monitoring and Visualization: An Exploratory Approach*. Springer, New York, 1999.

N. F. Johnson and S. Jajodia. Steganalysis: The investigation of hidden information. In *1998 IEEE Information Technology Conference*, pages 113–116, 1998.

N. F. Johnson and S. C. Katzenbeisser. A survey of steganographic techniques. In S. Katzenbeisser and F. A. P. Petitcolas, editors, *Information Hiding Techniques for Steganography and Digital Watermarking*, pages 43–78. Artech House, Boston, MA, 2000.

W.-H. Ju and Y. Vardi. A hybrid high-order Markov chain model for computer intrusion detection. Technical Report 92, National Institute of Statistical Sciences, 1999. Available at www.niss.org/downloadabletechreports.html.

M. H. Kang, J. N. Frocher, and I. S. Moskowitz. An architecture for multilevel secure interoperability. In *Proceedings of the 13th Annual Computer Security Applications Conference*, pages 194–204, 1997a.

M. H. Kang, A. P. Moore, and I. S. Moskowitz. Design and assurance strategy for the nrl pump. In *Proceedings of the 1997 High-Assurance Systems Engineering Workshop*, pages 64–71, 1997b.

M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network version of the pump. In *Proceedings of the 1995 Symposium on Security and Privacy*, pages 144–154, 1995.

M. H. Kang, I. S. Moskowitz, and D. C. Lee. A network pump. *IEEE Transactions on Software Engineering*, 22(5):329–338, May 1996.

S. Katzenbeisser and F. A. P. Petitcolas, editors. *Information Hiding techniques for steganography and digital watermarking*. Artech House, Boston, MA, 2000.

K. Kendall. A database of computer attacks for the evaluation of intrusion detection systems. Master's thesis, Massachusetts Instititute of Technology, 1999.

M. F. Kensey. Computer viruses - towards better solutions. *Computers and Security*, 12:536–541, 1993.

J. O. Kephart. A biologically inspired immune system for computers. In R. A. Brooks and P. Maes, editors, *Artificial Life IV. Proceedings of the 4th International Workshop on Synthesis and Simulation of Living Systems*, pages 130–139. MIT Press, Cambridge, MA, 1994.

J. O. Kephart, G. B. Sorkin, and M. Swimmer. An immune system for cyberspace. In *1997 IEEE International Conference on Computational Cybernetics and Simulation*, pages 879–884, 1997.

J. O. Kephart and S. R. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 343–359, 1991.

J. O. Kephart and S. R. White. Measuring and modeling computer virus prevalence. In *Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2–15, 1993.

J. O. Kephart, S. R. White, and D. M. Chess. Computers and epidemiology. *IEEE Spectrum*, 30(5):20–26, May, 1993.

K. M. Khalil, K. Q. Luc, and D. V. Wilson. LAN traffic analysis and workload characterization. In *Proceedings of the 15th Conference on Local Computer Networks*, pages 112–122, 1990.

D. V. Klein. Defending against the wily surfer - Web based attacks and defenses. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 81–92, 1999.

C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 268–279, 1994.

C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, 1997.

T. Kohonen. *Self-organizing maps*. Springer-Verlag, Berlin, 1995.

E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley & Sons, Inc., New York, eighth edition, 1999.

S. Kumar and E. H. Spafford. A generic virus scanner in C++. In *Proceedings of the Eighth Annual Computer Security Applications Conference*, pages 210–219, 1992.

T. Lane and C. E. Brodley. Sequence matching and learning in anomaly detection for computer security. In *AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk management*, 1997.

T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Computer Security*, 2:295–331, 1999.

J.-S. Lee, J. Hsiang, and P.-H. Tsang. A generic virus detection agent on the internet. In *Proceedings of the Thirtieth Hawaii International Conference on System Sciences*, pages 210–219, 1997.

W. Lee, C. T. Park, and S. J. Stolfo. Automated intrusion detection using NFR: Methods and experiences. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 63–72, 1999a.

W. Lee, S. J. Stolfo, and K. W. Mok. Data mining approaches for intrusion detection. In *Proceedings of the Seventh Usenix Security Symposium*, pages 79–93, 1998.

W. Lee, S. J. Stolfo, and K. W. Mok. A data mining framework for building intrusion detection models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 120–132, 1999b.

Y. K. Lee and L. H. Chen. High capacity image steganographic model. In *IEE Proceedings - Vision, Image and Signal Processing*, pages 288–294, 2000.

W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

D.-T. Lin. Computer-access authentication with neural network based keystroke identity verification. In *International Conference on Neural Networks*, pages 174–178, 1997.

U. Lindqvist and P. A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 1–16, May 1999.

R. F. Ling. A computer generated aid for cluster analysis. *Communications of the ACM*, 16(6):355–361, 1973.

R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogorod, R. K. Cunningham, and M. A. Zissman. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition, 2000*, volume 2, pages 12–26, 1999.

R. Lo, P. Kerchen, R. Crawford, W. Ho, J. Crossley, G. Fink, K. Levitt, R. Olsson, and M. Archer. Towards a testbed for malicious code detection. In *Compcon Spring '91 Digest of Papers*, pages 160–166, 1991.

P. Loshin. *TCP/IP Clearly Explained*. Academic Press, Boston, 1997.

P. Loshin. *Big Book of IPsec RFCs*. Morgan Kaufmann, San Diego, 2000a.

P. Loshin. *Essential Ethernet Standards: RFCs and Protocols Made Practical*. John Wiley & Sons, New York, 2000b.

T. F. Lunt. Real-time intrusion detection. In *Proceedings of the Thirty-Fourth IEEE Computer Society International Conference*, pages 348–353, 1989.

T. F. Lunt, R. Jagannathan, R. Lee, and A. Whitehurst. Knowledge-based intrusion detection. In *Proceedings of the Annual AI Systems in Government Conference*, pages 102–107, 1989.

T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, and C. Jalali. Ides: A progress report. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, pages 273–285, 1990.

L. K. Maisuria, C. S. Ong, and W. K. Lai. A comparison of artificial neural networks and cluster analysis for typing biometrix authentication. In *International Joint Conference on Neural Networks, IJCNN '99*, pages 3295–3299, 1999.

M. A. Maloof and R. S. Michalski. A method of partial-memory incremental learning and its application to computer intrusion detection. In *Proceedings of the Seventh International Conference on Tools with Artificial Intelligence*, pages 392–397, 1995.

D. J. Marchette. *The Filtered Kernel Density Estimator*. PhD thesis, George Mason University, 1996.

D. J. Marchette. A statistical method for profiling network traffic. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 119–128, 1999.

D. J. Marchette, C. E. Priebe, G. W. Rogers, and J. L. Solka. Filtered kernel density estimation. *Computational Statistics*, 11(2):95–112, 1996.

R. E. Marmelstein, D. A. V. Veldhuizen, and G. B. Lamont. A distributed architecture of an adaptive computer virus immune system. In *1998 IEEE International Conference on Systems, Man, and Cybernetics*, pages 3838–3843, 1998.

M. Martín-Bautista and M.-A. Vila. Building adaptive user profiles by a genetic fuzzy classifier with feature selection. In *The Ninth IEEE International Conference on Fuzzy Systems, 2000*, pages 308–312, 2000.

R. R. Martine. *Basic Traffic Analysis*. Prentice-Hall, Upper Saddle River, NJ, 1994.

L. M. Marvel and C. T. Retter. A methodology for data hiding using images. In *Military Communications Conference, MILCOM 98*, pages 1044–1047, 1998.

J. McAfee and C. Haynes. *Computer Viruses, Worms, Data Diddlers, Killer Programs, and Other Threats to Your System*. St. Martin's Press, New York, 1989.

G. McGraw and G. Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, pages 33–41, September/October 2000.

J. McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 lincoln laboratory evaluations. *ACM Transactions on Information System Security*, 3:to appear, 2000.

G. J. McLachlan and K. E. Basford. *Mixture Models: Inference and Applications to Clustering*. Marcel Dekker, New York, 1988.

G. J. McLachlan and T. Krishnan. *The EM Algorithm and Extensions*. John Wiley & Sons, New York, 1997.

G. J. McLachlan and D. Peel. *Finite Mixture Models*. John Wiley & Sons, New York, 2000.

L. Mé. GASSATTA, a genetic algorithm as an alternative tool for security audit trails analysis. In *First International Workshop on the Recent Advances in Intrusion Detection*, 1998. Web proceedings: www.zurich.ibm.com/~dac/Prog_RAID98/Table_of_contents.html.

C. Meadows and J. McLean. Computer security and dependability: Then and now. In P. Ammann, B. Barnes, and S. Jajodia, editors, *Computer Security, Dependability, and Assurance: From Needs to Solutions*, pages 166–170. IEEE Computer Society Press, New York, 1999.

C. P. Meinel. How hackers break in ... and how they are caught. *Scientific American*, pages 98–109, October, 1998.

J. Millen. 20 years of covert channel modeling and analysis. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 113–114, May 1999.

B. Miller. Vital signs of identity [biometrics]. *IEEE Spectrum*, 31(2):22–30, 1994.

E. L. Miller, D. Shen, J. Liu, and C. Nicholas. Performance and scalability of a large-scale n-gram based information retrieval system. *Journal of Digital Information*, 1(5), 2000. Available at www.cs.umbc.edu/cadip/pubs.html.

M. C. Minnotte and R. W. West. The data image: a tool for exploring high dimensional data sets. In *Proceedings of the ASA Section on Statistical Graphics*, 1998.

R. Morris and D. Lin. Variance of aggregated web traffic. In *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communication Societies*, pages 360–366, 2000.

I. S. Moskowitz and M. H. Kang. Covert channels – here to stay? In *COMPASS '94: Proceedings of the Ninth Annual Conference on Safety, Reliaility, Fault Tolerance, Concurrency and Real Time Security*, pages 235–243, 1994a.

I. S. Moskowitz and M. H. Kang. Discussion of a statistical channel. In *Proceedings of the 1994 IEEE-IMS Workshop on Information Theory and Statistics*, page 95, 1994b.

A. Mounji and B. L. Charlier. Continuous assessment of a Unix configuration: Integrating intrusion detection and configuration analysis. In *Proceedings of the 1997 Symposium on Network and Distributed System Security*, pages 468–472, 1997.

A. Mounji, B. L. Charlier, and D. Zampuniéris. Distributed audit trail analysis. In *Proceedings of the 1995 Symposium on Network and Distributed System Security*, pages 102–112, 1995.

B. Mukherjee, L. T. Heberlein, and K. N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, May–June, 1994.

C. Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, January, 1997.

I. Nåsell. On the time to extinction in recurrent epidemics. *Journal of the Royal Statistical Society B*, 61:309–330, 1999.

O. Nasraoui, R. Krishnapuram, and A. Joshi. Relational clustering based on a new robust estimator with application to web mining. In *Proceedings of the 18th International Conference of the North American Fuzzion Information Processing Society*, pages 705–709, 1999.

NCSC. A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center, November, 1993.

P. G. Neumann. *Computer Related Risks*. Addison-Wesley, New York, 1995.

P. G. Neumann and P. A. Porras. Experience with EMERALD to date. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 73–80, 1999.

R. E. Newman-Wolfe and B. R. Venkatraman. High level prevention of traffic analysis. In *Proceedings of the Seventh Annual Computer Security Applications Conference*, pages 102–109, 1991.

S. Northcutt. *Network Intrusion Detection. An Analyst's Handbook*. New Riders, Indianapolis, IN, 1999.

C. J. Nuzman, I. Saniee, W. Sweldens, and A. Weiss. A compound model for TCP connection arrivals. In *Proceedings of the ITC Specialist Seminar on IP Traffic Measurement, Modeling, and Management*, 2000. to appear. Also available at www.ee.princeton.edu/~cjnuzman/pubs/abs_tcp.html.

M. S. Obaidat and B. Sadoun. Verification of computer users using keystroke dynamics. *IEEE Transactions on Systems, Man, and Cybernetics*, 27(2):261–269, 1997.

V. Paxson. Emperically derived analytic models of wide-area TCP connections. *IEEE/ACM Transactions on Networking*, 2(4):316–336, August 1994.

V. Paxson and S. Floyd. Wide area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

C. E. Peláez and J. Bowles. Computer viruses. In *Proceedings of the Twenty-Third Southeastern Symposium on System Theory*, pages 513–517, 1991.

F. A. P. Petitcolas. Introduction to information hiding. In S. Katzenbeisser and F. A. P. Petitcolas, editors, *Information Hiding techniques for steganography and digital watermarking*, pages 1–14. Artech House, Boston, MA, 2000.

E. Petron. *Linux Essential Reference*. New Riders, Indianapolis, IN, 2000.

S. J. Phillippo. Practical virus detection and prevention. In *IEE Colloquium on Viruses and their Impact on Future Computing*, pages 2/1–2/4, 1990.

P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the National Information Systems Security Conference*, 1997.

C. E. Priebe and D. J. Marchette. Alternating kernel and mixture models. *Computational Statistics and Data Analysis*, 35:43–65, 2000.

P. E. Proctor. *The Practical Intrusion Detection Handbook*. Prentice-Hall, Englewood Cliffs, NJ, 2001.

J. O. Ramsay and B. W. Silverman. *Functional Data Analysis*. Springer, New York, 1997.

R. Riedi. An improved multifractal formalism and self-similar measures. *Journal of Mathematical Analysis and Applications*, 189:462–490, 1995.

J. A. Robinson, V. M. Liang, J. A. M. Chambers, and C. L. MacKenzie. Computer user verification using login string keystroke dynamics. *IEEE Transactions on Systems, Man, and Cybernetics*, 28(2):236–241, 1998.

J. A. Rochlis and M. W. Eichin. With microscope and tweezers: The worm from MIT's perspective. *Communications of the ACM*, 32:689–698, 1989. Reprinted in Denning [1990b].

M. Roughan, D. Veitch, and P. Abry. Real-time estimation of the parameters of long-range dependence. *IEEE/ACM Transactions on Networking*, 8(4):467–478, 2000.

P. Ryan and S. Schneider. *Modelling and Analysis of Security Protocols*. Addison-Wesley, London, 2001.

M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. Technical Report 95, National Institute of Statistical Sciences, 1999. Available at www.niss.org/downloadabletechreports.html, to appear in the February, 2001 issue of Statistical Science.

G. A. F. Seber. *Multivariate Observations*. John Wiley & Sons, Inc., New York, 1984.

C. Seife. Digital music safeguard may need retuning. *Science*, 290:917–919, November, 2000.

R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *USENIX Workshop on Intrusion Detection and Network Monitoring (ID '99) Proceedings*, pages 29–40, 1999a.

R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A high-performance network intrusion detection system. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 8–17, 1999b.

S. J. Shepherd. Continuous authentication by analysis of keyboard typing characteristics. In *European Contention on Security and Detection, 1995*, pages 356–359, 1995.

S. W. Shieh and V. D. Gligor. A pattern-oriented intrusion-detection model and its applications. In *1991 IEEE Computer Society Symposium on Research in Security and Provacy*, pages 327–342, 1991.

J. F. Shoch and J. A. Hupp. The "worm" programs - early experience with a distributed computation. In P. J. Denning, editor, *Computers Under Attack: Intruders, Worms, and Viruses*. Addison-Wesley, New York, 1990.

B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, New York, 1986.

G. J. Simmons. The history of subliminal channels. *IEEE Journal on Selected Areas in Communication*, 16(4):452–462, May, 1998a.

G. J. Simmons. Results concerning the bandwidth of subliminal channels. *IEEE Journal on Selected Areas in Communication*, 16(4):463–473, May, 1998b.

P. Simoneau. *Hands-On TCP/IP*. McGraw-Hill, New York, 1997.

S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, T. Grance, L. T. Heberlein, C.-L. Ho, K. N. Levitt, B. Mukherjee, D. L. Mansur, K. L. Pon, and S. E. Smaha. A system for distributed intrusion detection. In *Compcon Spring '91 Digest of Papers*, pages 170–176, 1991.

J. L. Solka, D. J. Marchette, and B. C. Wallet. Statistical visualization methods in intrusion detection. In *Proceedings of the* 32^{nd} *Symposium on the Interface: Computing Science and Statistics*, 2000.

A. Somayaji and S. Forrest. Automated response using system-call delays. In *USENIX*, 2000. Also available at www.cs.unm.edu/~forrest/isa_papers.htm.

A. Somayaji, S. Hofmeyr, and S. Forrest. Principles of a computer immune system. In *ACM New Security Paradigms Workshop*, pages 75–82, 1997.

E. H. Spafford. The Internet worm: Crisis and aftermath. *Communications of the ACM*, 32:678–687, 1989.

E. H. Spafford, K. A. Heaphy, and D. J. Ferbrache. A computer virus primer. In P. J. Denning, editor, *Computers Under Attack: Intruders, Worms, and Viruses*, pages 316–355. Addison-Wesley, New York, 1990.

E. H. Spafford and S. A. Weeber. Software forensics: Can we track code to its authors? *Computers and Security*, 12:585–595, 1993.

R. Spence. *Information Visualization*. Addison-Wesley, New York, 2001.

L. Spitzner. Passive fingerprinting, May, 2000. Available at the Web site www.enteract.com/~lspitz/papers.html.

N. Starr. Linear estimation of the probability of discovering a new species. *The Annals of Statistics*, 7(3):644–652, 1979.

W. R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, MA, 1994.

A. J. Stewart. Distributed metastasis: A computer network penetration methodology. Technical report, The Packet Factory, 1999. Available at www.packetfactory.net/Papers/index.html.

M. Stillerman, C. Marceau, and M. Stillman. Intrusion detection for distributed applications. *Communications of the ACM*, 42:53–61, 1999.

S. J. Stolfo, W. Fan, and W. Lee. Cost-based modeling for fraud and intrusion detection: Results from the JAM project. In *Proceedings of the DARPA Information Survivability Conference and Exposition, 2000*, volume 2, pages 130–144, 1999.

C. Stoll. *The Cuckoo's Egg*. Pocket Books, New York, 1990.

Swiss Academic & Research Network. Default TTL values in tcp/ip, 1999. Available at www.switch.ch/docs/ttl_default.html.

A.-H. Tan and C. Teo. Learning user profiles for personalized information dissemination. In *IEEE World Congress on Computational Intelligence*, pages 183–188, 1998.

K. M. C. Tan and D. S. Collie. Detection and classification of tcp/ip network services. In *Proceedings of the 13th Annual Computer Security Applications Conference*, pages 99–107, 1997.

C. Tebaldi and M. West. Bayesian inference on network traffic using link count data. *Journal of the American Statistical Association*, 93(442):557–572, 1998.

G. J. Tesauro, J. O. Kephart, and G. B. Sorkin. Neural networks for computer virus recognition. *IEEE Expert*, 11(4):5–6, 1996.

W. Theilmann and K. Rothermel. Dynamic distance maps of the internet. In *Proceedings of the Nineteenth Annual Joint Conference of the IEEE Computer and Communication Societies*, pages 275–284, 2000.

M. Theus and M. Schonlau. Intrusion detection based on structural zeros. *Statistical Computing & Graphics Newsletter*, 9:12–17, 1998.

H. Thimbleby, S. Anderson, and P. Cairns. A framework for modelling trojans and computer virus infection. *The Computer Journal*, 41(7):444–458, 1998.

K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8): 761–763, 1984. Reprinted in Denning [1990b].

D. M. Titterington, A. F. M. Smith, and U. E. Makov. *Statistical Analysis of Finite Mixture Distributions*. John Wiley & Sons, Inc., New York, 1985.

B. Toxen. *Real World Linux Security: Intrusion Prevention, Detection and Recovery*. Open Source Technology Series. Prentice-Hall, Englewood Cliffs, NJ, 2001.

E. R. Tufte. *The Visual Display of Quantitative Data*. Graphics Press, Cheshire, CT, 1983.

E. R. Tufte. *Envisioning Data*. Graphics Press, Cheshire, CT, 1990.

E. R. Tufte. *Visual Explanations*. Graphics Press, Cheshire, CT, 1997.

M. M. Van Hulle, editor. *Faithful Representations of Topographic Maps*. John Wiley & Sons, New York, 2000.

Y. Vardi. Network tomography: Estimating source-destination traffic intensities from link data. *Journal of the American Statistical Association*, 91(433):365–377, 1996.

B. R. Venkatraman and R. E. Newman-Wolfe. Transmission schedules to prevent traffic analysis. In *Proceedings of the Ninth Annual Computer Security Applications Conference*, pages 108–115, 1993.

G. Vert, D. A. Frincke, and J. C. McConnell. A visual mathematical model for intrusion detection. In *21st National Information Systems Security Conference*, 1998. available at csrc.nist.gov/nissc/1998/papers.html.

G. Vigna and R. A. Kemmerer. Netstat: A network-based intrusion detection system. In *Proceedings of the* 14^{th} *Annual Computer Security Applications Conference*, pages 7–11, 1998.

H. Wainer. *Visual Revelations: Graphical Tales of Fate and Deception from Mapolean Bonaparte to Ross Perot*. Springer-Verlag, New York, 1997.

G. G. Walter and M. Contreras. *Compartmental Modeling with Networks*. Birkhäuser, Boston, MA, 1999.

M. P. Wand and M. C. Jones. *Kernel Smoothing*. Chapman and Hall, London, 1995.

C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *1999 IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

M. S. Watterman, editor. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman & Hall/CRC, Boca Raton, 1995.

D. J. Watts. *Small Worlds*. Princeton University Press, Princeton, NJ, 1999.

E. J. Wegman. Hyperdimensional data analysis using parallel coordinates. *Journal of the American Statistical Association*, 95:664–675, 1990.

E. J. Wegman and D. B. Carr. Statistical graphics and visualization. In C. R. Rao, editor, *Handbook of Statistics 9: Computational Statistics*, pages 857–958. North Holland, Amsterdam, 1993.

E. J. Wegman, D. B. Carr, and Q. Luo. Visualizing multivariate data. In C. R. Rao, editor, *Multivariate Analysis: Future Directions*, pages 423–466. North Holland, Amsterdam, 1993.

J. D. Weiss and E. G. Amoroso. Ensuring software integrity. In *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, pages 323–330, 1988.

L. Wilkinson. *The Grammar of Graphics*. Springer, New York, 1999.

W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variablility: statistical analysis of ethernet LAN traffic and the source level. *IEEE/ACM Transactions on Networking*, 5:71–86, 1997.

G. J. Wills. Nicheworks - interactive visualization of very large graphs. *Journal of Computational and Graphical Statistics*, 8:190–212, 1999.

I. H. Witten. Computer (in)security: Infiltrating open systems. In P. J. Denning, editor, *Computers Under Attack: Intruders, Worms, and Viruses*. Addison-Wesley, New York, 1990.

X. Yang, A. P. Petropulu, and V. Adams. The extended on/off model for high-speed data networks. In *10th IEEE Signal Processing Workshop on Statistical Signal and Array Processing*, 1999.

A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 129–140, 1996.

R. L. Ziegler. *Linux Firewalls*. New Riders, Indianapolis, IN, 1999.

Glossary

Attacker A person who attacks a computer. See Cracker.

Back Door A section of code in a program or operating system that allows unauthorized access to those with the knowledge to exploit it. These are often put in by the original programmers to allow easy debugging in the early stages of development, but become a problem when the programmers forget (or choose not) to close them off.

Bias Bias is a statistical term which means the amount that the estimate differs from its expectation. For an estimate T of a quantity τ, the bias $b(T)$ is defined as $b(T) = E(T) - \tau$. For example, the sample mean is unbiased (has zero bias), since its expectation is the mean.

Cache Poisoning Changing the DNS cache on a machine so that a host (usually a Web server) is redirected to a different host.

Core Dump When a program dies unexpectedly, the operating system saves the current state of the program (memory values, registers, etc.) in a file (called "core"). A core dump, or "dumping core," refers to this process.

Covert Channel A channel of communications that is hidden. This can be done through placing data in unexpected places (such as packet header fields) or by encoding the message in subtle ways such as the timing of packet acknowledgments.

Cracker A person who attacks or gains access to a machine for malicious purposes.

Daemon A program that runs in the background; usually a system program that handles events such as connection requests or performs other maintenance functions.

Digraph A graph with directed edges.

DMZ The DMZ (demiliterized zone) consists of the machines on your network that lie outside your firewall. These may sit behind another firewall, but they are in some sense less protected than the rest of your network. They usually consist of those machines that the world is supposed to know about, such as Web servers, DNS servers, and mail servers.

Dòmain Name Server (DNS) A machine that maintains a database to perform the mapping between a host name and an IP address.

Dominating Set A set of vertices of a graph that are neighbors of all the vertices of the graph. The number of elements in a minimal dominating set is called the domination number.

Dumpster Diving See Trashing.

Ethernet A widely used networking technology.

Firewall Software that monitors and controls the communications into and out of a network (or machine).

Fork Create a child process. One also talks about "spawning" a process. The idea is to start up a new process (program) from an existing one.

Graph A graph is a set of vertices and edges between vertices. These are used to construct efficient data structures, to model various kinds of physical processes, and to construct efficient algorithms in computer science.

Hacker 1. A person who is a good and enthusiastic computer programmer. A prolific and expert coder. 2. A person who breaks into computers. People who adhere to definition # 1 call the latter "crackers."

Hamming Distance A distance measure that computes the number of places in which two strings differ.

Host A computer connected to a network.

Load Average A measure of the amount of work a computer is performing; the "load" on the computer.

Identifiable A set of parameters for a model are identifiable if they can be determined from the data in a way that uniquely determines the model. For example, if one has parameters a and b and can only determine $a + b$, the parameters are not identifiable since many choices for a and b result in the same sum.

Insider Threat An attacker that is a member of the attacked organization. This is usually a disgruntled employee or a prankster. Insiders are a particularly difficult and important problem since an insider by definition has access to the organization's computers.

Internet Service Provider (ISP) An organization that provides the "on ramp to the Information Superhighway" (sorry, I promise not to use that phrase again). An ISP provides modems, phone lines, or other infrastructure to allow computers to connect to its network (for example, from home via a modem).

Logic Bomb A computer program that appears innocuous until a respecified time, when it performs some nasty action, such as deleting files.

Malicious Code Any software written with evil intent. Examples include viruses, worms, and trojans.

Man Page The manual page for a command in the Unix operating system.

Mixture Model A model for the probability density function of a random variable as a convex sum of density functions.

Nearest-Neighbor Classifier An algorithm that assigns to new data the class associated with the nearest exemplar from a training set.

Outlier A datum that is in the tails of its (assumed) distribution. For example, when considering the daily rainfall on the island of Hawaii, the day (in 2000) when it received 38" of rain was an outlier.

Packet The fundamental information unit on a network. All communications are broken into packets, with each packet routed individually to the destination.

Patch Code to fix a bug or vulnerability in a program. Also used as a verb or adjective: to patch a machine means to install all appropriate patches; an unpatched machine is one that has known bugs that have not been corrected via the appropriate patches.

Port A number that UDP and TCP assign to network services and applications. Each application is assigned one or more ports, and these port numbers are then used to route packets to the appropriate application.

Probability Density Function (PDF) The continuous version of the probability mass function. One way to define it is as the derivative of the distribution function.

Probability Distribution Function Usually denoted with a capital letter, the probability distribution function returns the probability of observing a value at least as large as the one observed: $F(x) = P(X \leq x)$. See Probability Density Function.

Promiscuous A network interface in promiscuous mode will make a copy of every packet that passes the interface rather than only grabbing those packets destined for the interface.

Proxy A proxy is an application that acts as a gateway between two networks. It provides access control to the network and can also hide information about the protected network, hiding the internal network structure.

Red Team A group of "good guys" that plays "bad guys." The red team is used to try to break into, or otherwise attack, a system or network in order to determine the quality of the security or to identify weaknesses to be addressed.

Resolve To map an IP address to a machine name (or vice versa).

Root The "super user" or administrator on a Unix machine is called "root." This user has full permission for reading, writing, and executing essentially any file on the system. An attacker with root permission is said to "own" the machine since he or she can operate with impunity. A slang expression, "to get root on someone," has been coined as a result.

Rootkit A collection of programs that hides the activities of an attacker. This may include programs designed to give the attacker root permission on the

machine, change log files to eliminate evidence of the attack, and install trojan copies of system programs.

Router A machine that forwards packets on a network.

Script Kiddie An attacker who uses an attack script written by others. Usually, this is someone with little knowledge of the details of the attack, who is merely executing a program written by someone else.

Sensor A network sensor is a program that examines all the packets on a network interface and stores a subset of these packets for analysis.

Signature A pattern by which an entity or activity of interest can be identified.

Snail Mail Mail that goes through the postal system. This is to be contrasted with email, which gets delivered (nearly) instantaneously, assuming the mail servers are configured properly.

Sniffer A program that examines all the packets on a network interface. The distinction between a sensor and a sniffer is basically that network security officers use sensors, whereas everyone else uses sniffers.

Social Engineering Obtaining information (for example user IDs and passwords) by personal contact with someone who has the desired information. For example, the attacker calls the system administrator and poses as a vice president of the company who has forgotten his password and needs to get some data from his computer immediately for a meeting with an important client.

Spam Unsolicited email from strangers, the equivalent of junk mail or junk phone calls. This is often advertisement, sometimes part of a scam, and is generally detested.

Spawn See Fork.

Spoof To pretend to be something you are not. This is the term used when a packet is sent with a source IP address that has been changed to hide the identity of the true originator of the packet.

Stateful This refers to maintaining information about the state of a process. For example, a stateful firewall keeps track of the state of a TCP connection (whether the three-way handshake has been properly completed, for example) and denies connections that "break state."

Steganography The hiding of information in other information.

Super User The user with full permission on a Unix machine. This user (usually named "root") can execute, view, or modify any file on the computer and is in charge of maintaining, administering, and securing the computer.

TCP/IP Transmission Control Protocol/Internet Protocol. The lingua franca of the Internet. This is the protocol to which all packets on the Internet must conform, regardless of the specifics of their originating internal network.

Trashing Looking through trash in the hopes of finding useful information about an organization or individual. Attackers will do this in order to get user names, passwords, documentation, or other useful information.

Trojan A program that purports to perform one function but secretly performs another, usually a diabolical one.

Virus A computer virus is a program that copies itself into (infects) other programs. It may or may not perform other tasks. It can be passed by infected files on a disk, by files downloaded from another computer, or by attachments sent via email. See also the entry for Worm.

War Dialer A program that dials a range of telephone numbers looking for answering computers.

Worm A computer worm is a program that spawns running copies of itself, such as the Internet worm of 1988, which was one of the first and most famous of these computer denizens. An alternate definition used by some is that viruses require a user to propagate them (for instance by executing a program or inserting a disk) while worms do not.

Acronyms

ADC	Approximate Distance Clustering
AIC	Akaike Information Criterion
AIDE	Advanced Intrusion Detection Environment
AKA	Also Known As
AKMDE	Alternating Kernel and Mixture Density Estimation
BSM	Basic Security Module
CART	Classification and Regression Trees
CORBA	Common Object Request Broker Architecture
DARPA	Defense Advanced Research Projects Agency
DNS	Domain Name Service. Also, Domain Name Server
DOS	Denial of Service attack. Also, an old PC operating system, now generic for Microsoft operating systems.
DMZ	Demiliterized Zone
DDOS	Distributed Denial of Service
DOE	Department of Energy
DOS	Denial of Service
EMERALD	Event Monitoring Enabling Responses to Anomalous Live Disturbances
FAQ	Frequently Asked Questions
FKE	Filtered Kernel Estimator

FLD	Fisher's Linear Discriminant
FTP	File Transfer Protocol
GrIDS	Graph-Based Intrusion Detection System
GNU	GNU is Not Unix
HMM	Hidden Markov Model
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
IFS	Internet Field Separator
ILY	I Love You (a virus/worm)
IRC	Internet Relay Chat
ISP	Internet Service Provider
IP	Internet Protocol
LAN	Local Area Network
MAC	Media Access Control
MISE	Mean Integrated Squared Error
MITLL	Massachusetts Institute of Technology, Lincoln Labs
NCSC	National Computer Security Center
NFS	Network File Service
NIC	Network Information Center
NIDES	Next-generation Intrusion Detection Expert System
NSWC	Naval Surface Warfare Center
OS	Operating System
PC	Personal Computer
PD	Probability of Detection
PFA	Probability of False Alarm
PID	Process ID
PPP	Point-to-Point Protocol
RFC	Request For Comment
RIP	Routing Information Protocol
RIPPER	Repeated Incremental Pruning to Produce Error Reduction
ROC	Receiver Operating Characteristics
SANS	System Administration and Network Security
SETI	Search for Extra Terrestrial Intelligence
SHADOW	Secondary Heuristic Analysis for Defensive Online Warfare

SIR	Susceptible-Infected-Recovered
SIS	Susceptible-Infected-Susceptible
SRI	Sarnoff Research Institute
SSO	Site Security Officer
SWITCH	Swiss Academic & Research Network
TCP	Transmission Control Protocol
TFN	Tribe Flood Network
TFN2K	Tribe Flood Network 2000
TOCTTOU	Time-of-check-to-time-of-use
TTL	Time To Live
UDP	User Datagram Protocol
WAN	Wide Area Network
WATCHERS	Watching for Anomalies in Transit Conversation: a Heuristic for Ensuring Router Security

Author Index

Abry, P., 70, 291, 306
Adams, V., 310
Addie, R., 70, 291
Akaike, H., 138, 291
Ali, A., 216, 291
Allen, L., 297
Amoroso, E., xv, 147, 157, 255, 291, 310
Anderson, D., 171, 173–174, 177, 291
Anderson, E., 114, 291
Anderson, R., 255, 292
Anderson, S., 308
Andersson, H., 240, 292
Andrews, P., 210, 292
Anonymous, xv, 162, 292
Archer, M., 302
Ashmanov, I., 239, 292
Aura, T., 150, 299
Axelsson, S., 78, 84–85, 292

Bace, R., xv, 147, 157, 210, 292
Balasubramaniyan, J., 211, 292
Basford, K., 134, 303
Bass, T., 158, 292
Bauer, D., 211, 292
Bertin, J., 70, 292
Bestavros, A., 58, 294
Bharghavan, V., 9, 294
Bickel, P., 180, 292
Bishop, M., 150, 169–170, 292, 299
Bleha, S., 185, 293
Botstein, D., 296
Bowen, T., 306
Bowles, J., 240, 305
Brackney, R., 210, 293
Bradley, K., 150, 293
Breiman, L., 102, 293
Bretano, J., 307
Britton, T., 240, 292
Brodley, C., 188, 210, 301
Brown, M., 185, 293
Brown, P., 296
Browne, R., 244, 293
Büschkes, R., 210, 293

Cabrera, J., 80, 157, 293
Cairns, P., 308
Cao, J., 70, 293
Carr, D., 71, 310
Casey, E., 210, 293
Chambers, J., 306
Champion, T., 295
Chao, A., 180, 293
Chen, L., 247, 302
Cherukuri, R., 297
Chess, D., 301
Cheswick, B., 209, 293
Cheung, S., 150, 293–294
Chung, C., 211, 294
Cohen, F., 215, 220, 240, 294
Cohen, W., 182, 294
Collie, D., 157, 308
Comer, D., 42, 294
Contreras, M., 222, 309

Cowen, L., 133, 294
Craig, A., 48, 77, 299
Crawford, R., 294, 302
Criscuolo, P., 252, 294
Crossley, J., 302
Crovella, M., 58, 294
Cunningham, R., 302

Daley, D., 240, 294
Das, B., 9, 294
Davis, D., 293
Davis, R., 217, 294
Dempster, A., 51, 294
Denning, D., 163–164, 210, 220, 235, 242, 294–295
Denning, P., 210, 235, 239, 254–255, 295
Devroye, L., 77, 84, 295
D'haeseleer, P., 229, 295
Dias, G., 307
Di Battista, G., 58, 295
Dilger, M., 169–170, 292, 294
Dinwoodie, I., 70, 295
Dodge, M., 60, 295
Doverspike, R., 295
Duda, R., 84, 186, 295
DuMouchel, W., 196–197, 295, 306
Durbin, R., 190, 295
Durst, R., 81, 295

Eades, P., 58, 295
Eddy, S., 295
Eichelman, F., 292
Eichin, M., 235, 306
Eisenberg, T., 235, 296
Eisen, M., 115, 296
Elbaum, S., 210, 296
Endler, D., 210, 296
Escamilla, T., xv, 147, 161, 296
Ettema, D., 70, 157, 296
Everitt, B., 111, 113, 131–132, 296

Fan, W., 308
Farmer, D., 164, 296
Fedorov, V., 44–46, 296
Feldmann, A., 58, 70, 296–297
Ferbrache, D., 307
Finch, S., 180, 296
Fink, G., 301–302
Fiorini, P., 70, 297
Fisch, E., 210, 297
Fisher, R.A., 114, 297
Flanagan, D., 44–46, 296
Floyd, S., 53, 58, 80, 297, 305
Forrest, S., 178, 180, 182, 211, 230, 295, 297, 299, 307, 309
Foulds, L., 70, 297
Frank, J., 294
Freedman, D., 209, 297
Fried, D., 302
Friedman, J., 293
Frincke, D., 309
Frocher, J., 300
Fukunaga, K., 84, 297
Fyodor, 100–101, 297

Gani, J., 240, 294
Garber, L., 220, 237, 297
Garcia-Fernandez, J., 211, 292
Gentleman, R., 179, 299
Gertz, M., 294
Ghosh, A., 210, 297
Gilbert, A., 70, 296–297
Gilfix, M., 230, 298
Gilgert, A., 296
Gilham, F., 302
Gillespie, D., 293
Girardin, L., 157, 298
Gligor, V., 211, 307
Goan, T., 307
Goldberg, L., 231–232, 298
Goldberg, P., 298
Goldstein, E., 298
Graf, I., 302
Grance, T., 307
Green, J., 251, 298
Gries, D., 296
Guang, Y., 306
Györfi, L., 295

Hafner, K., 105, 163, 298
Haines, J., 302
Hall, E., 42, 298
Hand, D., 84, 298
Hart, P., 295
Hartmanis, J., 296
Haynes, C., 215, 235, 252, 303
Haynes, T., 9, 298
Heaphy, K., 307
Heberlein, T., 304, 307
Hedberg, S., 240, 298
Hedetniemi, S., 298
Helman, P., 210, 295, 298
Herodotus, 246, 298
Herrera, R., 292
Herringshaw, C., 157, 298
Hethcote, H., 240, 298
Highland, H., 215, 298–299
Hoagland, J., 294
Ho, C-L., 307
Ho, W., 302
Hofmeyr, S., 180–182, 230, 297, 299, 307
Hogg, R., 48, 77, 299
Holcomb, D., 296
Hruska, J., 240, 299
Hsiang, J., 301
Huang, P., 70, 296

Hughes, J., 150, 299
Hupp, J., 115, 239, 307
Hussien, B., 293

Icove, D., 210, 299
Ihaka, R., 179, 299
Ilgun, K., 210, 299
Inselberg, A., 66, 299
Irgon, A., 292

Jagannathan, R., 302
Jajodia, S., 246, 300
Jalali, C., 302
Jamil, T., 246, 299
Javitz, H., 171, 291, 299–300
Jeffrey, C., 210, 300
Johnson, N., 246–247, 300
Jones, M., 135, 137, 309
Joshi, A., 305
Ju, W-H., 197, 300, 306

Kang, M., 243–244, 255, 300, 304
Karr, A., 306
Kasperskaya, N., 239, 292
Katzenbeisser, S., 247, 249, 255, 300
Kemmerer, R., 157, 309
Kendall, K., 159, 165–167, 300, 302
Kensey, M., 240, 300
Kephart, J., 221–222, 226, 228–229, 300–301, 308
Kerchen, P., 302
Kesdogan, D., 293
Khalil, K., 70, 301
Kitchin, R., 60, 295
Klein, D., 26, 301
Ko, C., 167, 169, 210, 301
Kohonen, T., 157, 301
Kreyszig, E., 225, 301
Krishnan, T., 51, 134, 303
Krishnapuram, R., 305
Krogh, A., 295
Kumar, S., 218, 301
Kurtz, T., 296

Lai, W., 303
Laird, N., 294
Lamont, G., 303
Lane, T., 188, 210, 301
Le Charlier, B., 210, 304
Lee, D., 300
Lee, J., 240, 301
Lee, R., 302
Lee, W., 211, 301–302
Lee, Y., 247, 302
Leland, W., 56, 302
Levitt, K., 294, 301–302, 304, 307
Liang, V., 306
Liepins, G., 210, 298
Lin, D., 70, 185–186, 302, 304
Lindqvist, U., 211, 302
Ling, R., 115, 302
Lippmann, R., 81–82, 302
Liu, J., 304
Lo, R., 240, 302
Longstaff, T., 297
Loshin, P., 42, 302
Luc, K., 301
Lugosi, G., 295
Lunt, T., 209, 211, 291, 302
Luo, Q., 310
Lynn, M., 296

MacKenzie, C., 306
Maisuria, L., 185, 303
Makov, U., 308
Maloof, M., 211, 303
Mann, C., 209, 297
Mansur, D., 307
Marceau, C., 211, 308
Marchette, D., 126–127, 130–134, 136–137, 298, 303, 306–307
Markoff, J., 105, 163, 298
Marmelstein, R., 229–230, 303
Martín-Bautista, M., 211, 303
Martine, R., 70, 303
Marvel, L., 247, 303
McAfee, J., 215, 235, 252, 303
McClung, D., 302
McConnell, J., 309
McCulloch, R., 70
McGraw, G., 255, 303
McHugh, J., 84, 303
McLachlan, G., 51, 134, 303
McLean, J., 304
Meadows, C., 157, 304
Mé, L., 211, 304
Mehra, R., 293
Meinel, C., 157, 304
Mendell, N., 296
Michalski, R., 211, 303
Millen, J., 243, 304
Miller, B., 183, 304
Miller, E., 211, 295, 304
Minnotte, M., 115, 304
Mitchison, G., 295
Mok, K., 302
Moore, A., 300
Morris, R., 70, 304
Morrisett, G., 255, 303
Moskowitz, I., 243–244, 300, 304
Mounji, A., 210, 304
Mukherjee, B., 157, 293, 304, 307
Munson, J., 210, 296

Nachenberg, C., 219, 221, 304
Nasraoui, O., 211, 305
Neame, T., 291

Neumann, P., 146–147, 157, 210, 302, 305–306
Newman-Wolfe, R., 157, 244, 305, 309
Nicholas, C., 304
Northcutt, S., xv, 89, 105, 298, 305
Nåsell, I., 240, 305
Nuzman, C., 53, 305

Obaidat, M., 185–186, 293, 305
Olshen, R., 293
Olsson, R., 302
Ong, C., 303

Park, C., 301
Paxson, V., 53, 55–56, 58, 80, 297, 305
Pearlmutter, B., 309
Peel, D., 134, 303
Peláez, C., 240, 305
Perelson, A., 297
Peterson, M., 210, 292
Petitcolas, F., 246–247, 249, 255, 292, 300, 305
Petron, E., 42, 305
Petropulu, A., 310
Phillippo, S., 221, 305
Phillips, C., 298
Pon, K., 307
Pooch, U., 297
Porras, P., 146–147, 157, 211, 302, 305–306
Priebe, C., 133–134, 294, 303, 306
Proctor, P., 74, 210, 306
Puketza, N., 293

Ralph, B., 298
Ramsay, J., 145, 171, 306
Ravichandran, B., 293
Reichl, P., 293
Retter, C., 247, 303
Riedi, R., 70, 306
Robinson, J., 186, 306
Rochlis, J., 235, 306
Rogers, G., 303
Rogers, S., 185, 293
Rothermel, K., 70, 308
Roughan, M., 70, 306
Rowe, J., 294
Rubin, D., 294
Ruschitzka, M., 301
Ryan, P., 158, 306

Sadoun, B., 185–186, 305
Saniee, I., 295, 305
Santoro, T., 296
Schartzbard, A., 297
Schatz, M., 297
Schneider, S., 158, 306
Schonlau, M., 188, 192–193, 195, 197–200, 295, 306, 308
Seber, G., 51, 306
Segal, M., 306
Seife, C., 247, 306
Sekar, R., 157, 209, 306
Shanbhag, T., 306
Shen, D., 304
Shepherd, S., 183, 307
Sherman, R., 310
Shieh, S., 211, 307
Shoch, J., 115, 239, 307
Silverman, B., 135, 137, 145, 171, 306–307
Simmons, G., 243, 255, 307
Simoneau, P., 42, 307
Slater, P., 298
Slivinsky, C., 293
Smaha, S., 307
Smith, A., 308
Snapp, S., 211, 307
Solka, J., 114, 303, 307
Somayaji, A., 182, 297, 299, 307
Sorkin, G., 298, 300
Spafford, E., 218, 232, 235, 239, 301, 307
Spagnuolo, L., 295
Spellman, P., 296
Spence, R., 71, 307
Spitzner, L., 307
Staniford-Chen, S., 294
Starr, N., 180, 308
Stevens, W.R., 3, 5, 9, 15, 19, 23–24, 42, 308
Stewart, A., 157, 308
Stillerman, M., 211, 308
Stillman, M., 308
Stolfo, S., 211, 301–302, 308
Stoll, C., 209, 308
Stone, C., 293
Stork, D., 295
Sweldens, W., 305
Swimmer, M., 300

Tamaru, A., 291, 302
Tamassia, R., 58, 295
Tan, A., 211, 308
Tan, K., 157, 308
Taqqu, M., 302, 310
Tebaldi, C., 46, 70, 308
Teo, C., 211, 308
Tesauro, G., 240, 308
Theilmann, W., 70, 308
Theus, M., 195, 306, 308
Thimbleby, H., 255, 308
Thode, H., 296
Thompson, K., 254, 308
Timmermans, H., 70, 157, 296
Titterington, D., 134, 308

Tollis, I., 58, 295
Toxen, B., 10, xv, 309
Tsang, P., 301
Tufte, E., 71, 309

Valdes, A., 171, 291, 299–300
Vander Wiel, S., 293
Van Hulle, M., 157, 309
Van Veldhuizen, D., 303
Vardi, Y., 46–49, 51–52, 70, 197, 300, 306, 309
Veitch, D., 70, 291, 306
Venema, W., 164, 296
Venkatraman, B., 157, 244, 305, 309
Verma, S., 306
Vert, G., 209, 309
Vigna, G., 157, 309
Vila, M., 211, 303

Wainer, H., 71, 309
Wallet, B., 307
Walter, G., 222, 309
Wand, M., 135, 137, 309
Warrender, C., 180, 182, 309
Watterman, M., 190, 309
Watts, D., 71, 309
Weber, D., 302
Webster, S., 302
Webster, W., 304
Weeber, S., 232, 307
Wegman, E., 66, 71, 309–310
Weiss, A., 305
Weiss, J., 255, 310
Wenke, L., 308
West, M., 46, 70, 308
West, R., 115
White, G., 297
White, S., 222, 226, 228, 301
Whitehurst, A., 302
Wilkinson, L., 71, 310
Willinger, W., 58, 296–297, 302, 310
Wills, G., 58, 310
Wilson, D., 301–302, 310
Witten, B., 295
Witten, I., 254, 310
Wyschogorod, D., 302

Yahav, J., 180, 292
Yang, X., 58, 310
Yip, R., 294
Young, A., 240, 310
Yu, B., 293
Yung, M., 240, 310

Zampuniéris, D., 304
Zerkle, D., 294
Ziegler, R., 208–209, 310
Zissman, M., 302
Zukerman, M., 291

Subject Index

.cshrc, 199
.rhosts, 165–166
/bin/sh, 170
/etc/hosts.deny, 205
/etc/passwd, 9, 35, 163, 169
/etc/services, 155
/etc/tw.config, 206
/ftp/etc, 164
/tmp, 161, 169–170

Acknowledgment number, 19–20, 105–106
Active sensor, 44
Activity graph, 150–151
Activity profiling, 109, 163, 210
Acyclic graph, 231
ADC, 133–134, 140–144, 317
Agent, 157, 211, 252–253
AIC, 138, 317
Aide, 206, 317
Akaike information criterion, 138
AKMDE, 134–135, 139, 317
Alias, 162, 199
Anomaly detection, 150, 210, 229, 238, 251
Anomaly detector, 150
Anonymity, 157
Anonymous FTP, 165
Anti-virus software, 219
Apache Web server, 160
Application layer, 5
Arrival times, 53
Attacks
  covering up, 171
  DDOS, 252
    Stacheldraht, 253
    TFN, 253
    TFN2K, 253
    Trin00, 253
  denial-of-service, 234, 252
  DOS, 252
  host
    apache2, 160
    back, 160
    buffer overflow, 165–167
    denial-of-service, 159
    FTP write, 165
    loadmodule, 166
    mailbomb, 160
    password cracking, 163
    password guessing, 163
    phf, 35, 164
    race condition, 168
    remote to user, 159
    resource hogging, 161
    spam, 160
    trojans, 166
    user to root, 159
    webbomb, 161
    Windows NT, 162
  network, 91
    denial-of-service, 91, 157

gaining access, 103
hijacking, 100, 153
icmp flood, 253
land, 91–93
neptune, 92
network mapping, 98
password guessing, 104
ping of death, 93
probe, 98, 102
process table, 93
smurf, 95–96, 253
SYN flood, 253
syslogd, 95
targa3, 94, 253
TCP hijacking, 105
teardrop, 95
UDP flood, 253
UDP storm, 96
remote to user, 162
user to root, 166
Audit record, 174
Autonomous agent, 211

Back door, 241, 254, 311
Back orifice, 250
Bacon, Kevin, 71
Base-rate fallacy, 78
Bayes' theorem, 77
Beginning of time, 7
Bias, 75, 311
Big endian byte order, 13
Biometric, 183, 200
Bourne shell, 170, 200
BSM, 204, 210, 317
Buffer overflow attack, 180
Buffer overflow, 165–167

Cache poisoning, 26, 311
CART, 102, 317
Chargen port, 97
Checksum, 14, 17, 19, 206, 218, 244, 246
Chmod, 170
Chown, 170
Classifier, 74, 185
k-fold cross validation, 76
k-nearest neighbor, 77, 185, 187
k-point cross validation, 76
1-point cross validation, 76
CART, 102
cross validation, 76
evaluation, 75
IPAM, 197
linear, 185
Markov model, 197
minimum distance, 185–186
nearest-neighbor, 77, 79, 185–188, 313
neural network, 157, 185–186, 240
quadratic, 185
resubstitution, 77
selection bias, 75
test set, 75–76
training set, 75
Clustering
k-means, 131–132, 134
agglomerative, 115
approximate distance, 133
complete linkage, 115
hierarchical, 111, 113, 115–116
nearest-neighbor, 111
Cohen, William, 182
Color histogram, 68–69, 114–115
Common user names, 163, 165
Compartmental model, 222
Compression, 108, 197, 219, 247, 249
Computer immunology, 178
Confusion matrix, 187
CORBA, 211, 317
Core dump, 94, 311
Cornell, 235
Correlation, 149
Country code, 24, 276–279
Cover, 247
Covert channel, 15, 242–243, 255, 311
Covert_tcp, 243
Crack, 164
Cracker, 311–312
Cross validation, 76, 187
Cshell, 161, 200, 232
Cult of the Dead Cow, 250
Cybergeography, 60

D-optimal experimental design, 46
Daemon, 311
DARPA, 79, 81–82, 93, 317
Data encapsulation, 10
Data fusion, 158
Data image, 114–119, 188–189, 191–194
Data mining, 211
Datagram, 16
DDOS, 252, 317
Death by monoculture, 216
Deep throat, 250
Default gateway, 23
Dendogram, 113, 115, 118, 189
Denial-of-service, 78, 91–96, 100, 106, 160–162, 206, 234, 238, 252
Density estimation
AKMDE, 134
filtered kernel estimator, 136, 138
FKE, 136
kernel estimator, 135
nonparametric, 135–136
DF, 7, 21
Diameter of the Internet, 102
Digraph, 46, 223, 231, 312

Directed graph, 46, 231
Dirichlet distribution, 196
Distributed denial-of-service, 252
Distribution
  χ^2, 177
  exponential, 56
  extreme, 56
  gamma, 55
  log-extreme, 56
  lognormal, 56
  normal, 134–135
  Pareto, 56
  Poisson, 48, 51–53, 57, 70, 80
DMZ, 312, 317
DNA, 190
DNS server, 252
DNS, 4, 23–25, 53, 90, 95, 120, 122, 312, 317
  cache poisoning, 26
DOE, 317
Doly trojan, 250
Domain name server, 4, 8, 23–24, 312
Dominating set, 9, 312
Domination, 9
DOS, 317
Dumpster diving, 163, 312

Echo port, 97
EM algorithm, 50, 134
  normal mixture, 134
Email attachment, 237
Email virus, 235
Email, 53, 56, 64, 66, 79–80, 82–83, 104, 121, 145, 160–161
EMERALD, 126, 146, 150, 157, 201, 317
  analysis unit configuration, 149
  communications, 149
  engine configuration, 149
  event collection, 149
  event structures, 149
  generic monitor, 149
  profiler, 149
  resolver, 149
  resource object, 149
  response methods, 149
  signature engine, 149
  subscription lists, 149
Encryption, xiv, 29–30, 149, 158–159, 163, 168, 206, 218–219, 240, 244–245, 253
Enterprise-wide, 108
Epidemic, 222
Epidemiology, 221
Ethernet, 312
Euclidean distance, 185, 190–191
Evaluation
  live network testing, 82
  ROC curve, 79
Execution stack, 166–167
Experimental design, 44–46
Extinction rate, 222

File integrity checker, 165, 170, 206
File integrity, 218
Filtered kernel estimator, 137
Finger, 94, 122, 167, 235
Fingerprinting, 100, 151, 155–156
Finite-state machine, 157, 211
Firewall, 107, 207–208, 238, 250–251, 312
Fisher's linear discriminant, 186
FKE, 135, 317
FLD, 186, 318
Fork bomb, 232
Fork, 312
Forrest, Stephanie, 180, 197, 230
Fping, 28
Fragmentation, 21, 93
FTP, 10, 53, 56, 82, 92, 104, 120–121, 127, 129, 163–165, 200, 238, 318
Fyodor, 100–101

Gateway, 22
Gcc, 170, 179–181
Genetic algorithm, 211
Gets, 167
Gnu, 170
GNU, 318
Graph theory, 9, 46, 71
Graph, 9, 46, 58, 150, 231, 239, 312
  degree, 239
  digraph, 223
  random, 223
  size, 223
GrIDS, 150, 239, 318
Gzip, xvi, 108

Hacker, 312
Half-open scan, 151
Hamming distance, 181, 312
Happy 99, 252
Hardware layer, 4–5
Herodotus, 246
Hidden files, 171
Hidden Markov Model, 182
Highland, Harold, 240
HMM, 182, 318
Hofmeyr, Steven, 230
Home network, 6
Host, 312
Http, 6, 10, 160
Https, 123
Human genome project, 115
Human in the loop, 75

I Love You, 237, 318

IANA, 318
IBM, 240
ICMP, 7, 15, 22–23, 27, 30–31, 33, 35, 50, 93, 96, 98, 100–101, 208, 243, 249, 251, 253, 318
  destination unreachable, 101
  echo reply, 15–16, 96, 249, 253
  echo request, 15–16, 93, 98, 249
  header, 15
    code, 15
    type, 15
  port unreachable, 30
Identifiable, 48, 312
IDES, 209
IDS, 78, 150, 318
Ifconfig, 36
IFS, 318
Imap, 165
Immune system, 182
Immunology, 178, 216, 219, 229
Inappropriate usage, 9
Inetd, 251
Infection rate, 223
Insider threat, 200, 312
Interarrival times, 53–55
Internal field separator, 166
Internet Mapping Project, 58
Internet service provider, 7, 24, 131, 312
Internet weather report, 60
Internet worm, 234–235
Internet, 4, 15, 38, 50, 58, 60, 71, 80, 102, 157, 221, 229, 235, 253
Interpoint distance matrix, 116, 118–119, 125, 191–193
Interval-based IDS, 107
IP, 7, 11, 318
  address, 4, 6–7, 14–15, 17, 23–24, 26, 28–31, 33–34, 37, 40
  don't fragment flag, 101
  fragmentation, 21
  header
    checksum, 14
    destination IP address, 14
    don't fragment flag, 102, 156
    fragment offset, 14
    identification, 14
    IP flags, 14
    options, 14
    protocol, 14
    source IP address, 14
    time-to-live, 14, 102
    total length, 13
    ttl, 102
    type of service, 13, 102
    version, 13
  layer, 4–5
  options, 14
    loose source routing, 14
    record route, 14
    strict source routing, 15
    timestamp, 14
Ipchains, 207
IRC, 238, 318
Iris data, 114
ISP, 24, 161, 312, 318

Johns Hopkins, 180
JPEG, 238
Junk email, 160

Kephart, Jeffrey, 240
Kernel estimator, 135
  choice of bandwidth, 135, 137
Kevin Bacon Game, 71
Keystroke timing, 183–188, 200
Kill, 41, 232
Kim, Gene, 206

LAN, 318
Latency, 60
Lincoln labs, 79
Linear predictor, 44
Linux, xv, 37, 42, 82, 101, 107, 156, 178, 204, 207, 238, 253–254
Little endian byte order, 13
Load average, 160
Local area network, 10
Log server, 171
Logcheck, 204, 206
Logger, 129
Logic bomb, 252, 312
Login, 254
Loki, 15, 243, 249–250
Looping, 161, 163
Loose source routing, 14
Low-and-slow scan, 98
Ls, 171, 253
Lsof, 39, 251

MAC, 10, 318
Machine-learning, 182, 188, 211
Macro, 236
Mailbomb, 146
Malicious code, 94, 108, 234, 255, 313
Man page, 313
Marcum, Ron, 250
Markov model, 180, 196–197
Maximum segment size, 156
Mean integrated squared error, 135, 137
Melissa, 234, 236–237
Memory, 94
MF, 21
Michaelangelo effect, 228
Microsoft Windows, 215, 250
Microsoft, 215, 237–238, 317
Minimum distance classifier, 185

MISE, 137, 318
MIT, 79, 244
MITLL, 79–80, 82, 93, 318
Mitnick, Kevin, 105
Mixture model, 134–135, 313
  AKMDE, 134
  number of components, 135
Modem, 38, 162
Monte Carlo, 70
Monty Python, 241
Morris, Robert, 235
MS-DOS, xvi–xvii, 215
Multilevel security, 242–243, 255
Muuss, Mike, 27

N-gram, 178–181, 211, 229
Naiman, Dan, 180
Named, 165
NCSC, 243, 318
Netbus, 250
Netscape, 30, 241
Netstat, 36–37, 171, 251
Network
  byte order, 13
  layers, 5
  mapping, 78, 98
  modeling, 53
  profiling, 109
  pump, 243–244
  tomography, 46, 49, 70
  traffic intensities, 43, 70, 80
  traffic, 53, 58
Neural network, 210
News, 53, 56
NFS, 39, 127, 318
NIC, 318
NIDES, 149, 171, 201, 209, 318
Nmap, 107, 110, 151, 156, 239, 250
Nonself, 230
Normal activity, 110
Normal density, 134–135
Normal distribution, 134–135
Normal mixture, 134–135
Novel attacks, 81
Nslookup, 28–29, 107
NSWC, 9, 25, 80, 89, 106, 109, 163, 318
Null scan, 155

Octet, 4
On/Off model, 58
Operating system fingerprinting, 100, 151, 156
Outlier detection, 117–119, 132
Outlier, 313

P0f, 155–156
Packet, 313
Pairs plots, 63–64, 66–67, 114
Parallel coordinates, 66, 68, 114
Passive fingerprinting, 155–156
Passive sensor, 44
Password, 8, 11, 26, 29, 104, 112, 163–166, 168–169, 183–184, 186, 206, 218, 235, 238, 241, 249, 254, 314
  /etc/passwd, 169
  authentication, 186
  cracking, 163, 166
  file, 164–166, 168–169, 206, 235, 249
  guessing, 104, 163
  shadow, 165
  sniffing, 166
Patch, 313
Pattern recognition, 84
PD, 73, 318
PDF, 313
Perl, 106
Personal computer, 220
PFA, 73, 318
Phf, 35, 164
PID, 203, 318
Ping, 15, 27–28, 46, 93
Point to point protocol, 37–38
Poisson distribution, 48, 51–53, 70, 80
Port numbers, 6, 257–263
Port scan, 155, 250
Port, 313
Portal of doom, 250
Portsentry, 49, 205
Postal analogy, 3
PPP, 37–38, 318
Pppstats, 38
Precision, 74
Privilege, 166
Probability density function, 50, 196, 313
Probability distribution function, 313
Probability of detection, 73
Probability of false alarm, 73
Probe, 78, 98
Process ID, 41, 179, 203, 233
Process monitoring, 209
Profiler, 149
Profiling database users, 211
Promiscuous, 7, 36, 129, 313
Protocol layer, 4–5, 106
Protocols
  ICMP, 7, 15, 33
  Internet control message protocol, 15
  Internet protocol, 11
  IP, 7, 11
  PPP, 10
  RIP, 23
  routing information protocol, 23
  TCP, 7, 10, 17, 19, 33
  transmission control protocol, 17

UDP, 7, 10, 16, 33
user datagram protocol, 16
Proxy, 313
Ps, 171, 203
Purdue University, 206

R, 64, 66, 179
Race condition, 168, 209
Ramen, 238
Random graph, 216, 222
Rdist, 169–170
Read-only interface, 108
Real-time IDS, 107
Recall, 74
Receiver operating characteristic curve, 74
Record route, 14
Recursive formulas, 172
Red Hat Linux, 31, 42, 179
Red team, 83, 313
RESET packets, 251
Resolve, 313
RFC, 84, 102, 318
Ringzero, 250
RIP, 23, 318
RIPPER, 182, 211, 318
Rivest, Ronald, 244
Rlogin, 104, 165, 200
Rm, 162
ROC curve, 74, 79, 130, 142
ROC, 318
Root DNS servers, 26
Root, xvii, 162, 166, 313–314
Rootkit, 204, 253, 313
Router, 4–5, 314
Routing matrix, 47–48
Routing table, 22, 49
Routing, 22

S, 179
Saint, 239
SANS, 250, 318
Scatter matrix, 186
Scatter plots, 60–63
Scp, 29–30
Script kiddie, 167, 314
Secure channel, 29
Secure shell, 29, 107–108
Security policy, 7
Security tools
aide, 206
ifconfig, 36
ipchains, 207
logcheck, 204
lsof, 39, 204
netstat, 36
NIDES, 171
nmap, 151
nslookup, 28
p0f, 156
ping, 27
portsentry, 205
pppstats, 38
ps, 203
route, 37
SHADOW, 106
snort, 33
ssh, 29
swatch, 205
tcpdump, 6
tcpshow, 31
top, 203
traceroute, 30
tripwire, 165, 206
whois, 29
Selection bias, 75
Self-similar, 56–58, 70
Sendmail, 165, 182, 235
Sensor, 107, 314
Sequence number, 6, 19–20, 35, 105, 153, 243–246, 250
SETI, 234, 318
SHADOW, 25, 29–30, 33, 75, 80, 89, 106–109, 130, 149, 318
Shell, 200
Shimomura, Tsutomu, 105
Signature, 149, 168, 219, 314
Silicon Graphics, 104
SIR, 240, 319
SIS, 222, 319
Site security officer, 78
Small world, 71
Snail mail, 160, 314
Sniffer, 6, 8, 29–30, 33, 49, 314
Snort, 33–34, 36, 109, 149
Social engineering, 104, 163, 166, 314
Socket, 39
Sockets de troie, 250
Software patch, 216–217
Solaris, 82, 204, 253
Spafford, Eugene, 206
Spam, 146, 160, 314
Spawn, 312, 314
Spoof, 26, 49, 314
Spoofing, 91, 95–96, 98, 100, 105, 157, 160–161, 251, 253, 314
SRI, 146, 319
Ssh, 27, 29–30, 90, 104, 121, 123, 200, 208
SSO, 78, 83, 319
Stacheldraht, 252
Stateful firewall, 99, 314
Stateful, 243, 314
Statistical anomaly, 149
Steganography, 242, 246, 248–249, 314
Strace, 178
Strict source routing, 15

Strings, 201, 254
Su, 166
Subseven, 250
Suid, 170
Sun, 204
SunOS, 82
Super user, 159, 166, 313–314
Swatch, 205
SWITCH, 102, 319
Switched network, 8
SYN flood, 253
SYN packet, 7, 19, 69, 91, 93, 130, 209, 251

TCP, 6–7, 10, 17, 20, 33, 50, 53, 93, 98, 106, 154, 161, 205, 243, 246, 253, 313, 319
  checksums, 246
  connections, 19
  FIN packet, 101
  header
    acknowledgment flag, 19
    acknowledgment number, 18
    checksum, 19
    destination port, 18
    FIN flag, 19
    flags, 18
    length, 18
    options, 19
    PUSH flag, 19
    reserved, 18
    reset flag, 19
    sequence number, 18
    source port, 18
    SYN flag, 19
    urgent flag, 18–19, 102
    urgent pointer, 19, 102
    window size, 19, 102, 156
  options, 7, 101
    maximum segment size, 156
  reset packet, 99–100, 155
  scan, 154
  sequence number, 101
  SYN flag, 151, 155
  SYN packet, 7, 91–93, 99, 105–106, 110, 126, 130, 156, 251
  SYN scan, 151
  three-way handshake, 19–20, 50, 53, 68, 105, 154, 161, 314
TCP/IP, 314
Tcpdump, 6, 8, 33–34, 36, 49, 80, 89–90, 106, 108–109, 129, 153, 205
  filters, 90
Tcpshow, 31
Telnet, 56, 77, 80, 82, 90, 92, 104, 121, 162–163, 200, 241, 243, 246
Test data, 82
Test
  χ^2, 56
  Kolmogorov-Smirnov, 80, 157
TFN, 252–253, 319
TFN2K, 252–253, 319
TFTP, 164–165
Time-to-live, 14, 30–31, 156
Timeout, 92
Timestamp, 14
TOCTTOU, 169, 319
Top, 203
Traceroute, 27, 30–31, 50, 81, 102, 109, 127, 130
Traffic analysis, 157
Training data, 82
Transition probability, 196
Transmission control protocol, 17
Trap door, 249
Trashing, 163, 314
Tribe Flood Network, 252
Trin00, 252–253
Tripwire, 165, 170, 206
Trojan Horse, 241
Trojan port numbers, 266–274
Trojan, 38, 166, 171, 241–242, 254, 314
Trojans, 100, 110, 166, 252, 265
  back door, 249
  back orifice, 250
  deep throat, 250
  doly trojan, 250
  happy 99, 252
  Loki, 15, 249–250
  netbus, 250
  portal of doom, 250
  ringzero, 250
  sockets de troie, 250
  subseven, 250
TTL, 14, 31, 102, 156, 319
Type I error, 73, 185–186
Type II error, 74, 185–186
Type of service, 102

UC Davis, 150
UDP, 7, 10, 16, 19–20, 26, 30–31, 33–34, 90, 97, 126, 128, 130, 155, 205, 253, 313, 319
  datagram, 16
  header
    checksum, 17
    destination port, 17
    length, 17
    source port, 17
Unauthorized use, 230
Unix, 7, xv–xvii, 82, 93, 163, 171, 220, 232–233, 238, 253–254
  commands, xvi
Unix
  tools
    nslookup, 28

ping, 27
ps, 203
strings, 201
top, 203
Usenet, 104
User authentication, 183, 185
User profile, 211
User profiling, 163, 183, 210
User session, 53

Virus, 83, 215–216, 221, 242, 250, 252, 315
detection software, 219
detection, 218, 220, 230
generic decryption, 219
hoax, 220
immunology, 229
macro, 220, 236
Melissa, 220
metastable distribution, 226
metastable phase, 226
Michaelangelo effect, 228
phylogeny, 231
polymorphic, 219
propagation, 216
replication, 216–218
scanners, 218
signature, 219
SIS model, 222
Visualization
color histogram, 68, 114, 132
data image, 114, 117, 188
pairs plots, 63, 114
parallel coordinates, 66, 114
scatter plots, 60

WAN, 319
War dialer, 315
War dialing, 163
WATCHERS, 150, 319
Web resources, 281
Web server, 160
Web, 10, 30, 53, 62–63, 70–71, 79–80, 82, 99, 104, 120, 122–123, 129, 160–161, 164, 211, 242
Wegman, E., 68
Whois, 27, 29, 107
Window size, 7, 102, 156
Worm, 94, 115, 151, 161, 216, 220–221, 232, 235, 242, 252, 315
ILY, 237
macro, 236
melissa, 83, 236
ramen, 238

X Windows, 30, 36, 127, 166
Xerox, 215
Xhost, 166
Xlock, 166
Xmas scan, 155

Yertle the Turtle, 168
Yugoslavia, 58