Approaches to Computer Security:
Filtering, Testing, and Detection

by

NICHOLAS JOSEPH PUKETZA

B.S. (Stanford University) 1988
M.S. (University of California, Davis) 1996

DISSERTATION
Submitted in partial satisfaction of the requirements for the degree of
DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

Professor Biswanath Mukherjee (Co-Chair)

Professor Ronald A. Olsson (Co-Chair)

Professor Matthew A. Bishop

Committee in Charge

2000

Abstract

In recent years a variety of approaches to computer security have evolved, including filtering, testing, and detection. In this dissertation, we present research based on each of these approaches.

We first introduce a design and prototype for a device called Safe Modem that protects a computer from Internet threats. Safe Modem examines network packets and discards suspicious ones. We implemented two key features in the prototype: e-mail filtering and header-based packet filtering. We also conducted experiments to measure communication delays caused by the prototype.

Regarding testing, we present a methodology for testing intrusion detection systems (IDSs). The methodology includes test-case selection strategies and detailed testing procedures, especially for stress-testing. We include quantitative results from testing an IDS called Network Security Monitor. We also describe a software platform we developed to support the methodology.

Concerning detection, we present a protocol called WATCHERS that detects routers that drop or misroute packets. The protocol checks for "conservation of flow" in each router: the number of data bytes in packets flowing into a router should match the number of data bytes in packets flowing out. We discuss WATCHERS' response to several different attack scenarios. We also provide a complexity analysis of WATCHERS' memory and communication requirements and running time.

We next introduce a filtering strategy that protects network programs against message-flooding attacks. Under this strategy, the server scans its queue of incoming mes-

sages periodically and discards messages from unauthorized or misbehaving clients. The defense depends on a strong message-authentication method for identifying unauthorized clients. We present simulation results showing that the strategy helps a server to perform significantly better during a flooding attack. We also present a preliminary mathematical model of a server using the strategy.

As a second example of testing research, we introduce a methodology for testing network programs, based on analyzing the grammar that specifies valid incoming messages. The methodology is an instance of a general testing approach called *partition testing.* We describe how we used the methodology to test the Unix finger daemon program.

Contents

List of Figures

List of Tables

Acknowledgements

I've had the good fortune to be surrounded by many good people during my graduate school years.

I've always appreciated the friendly atmosphere in the Computer Science Department. The people of the Main Office have always treated me kindly, and have helped me countless times. Thank you to Mary Brown, Debbie Chadwick, Melinda Day, Michele Fulton, Meshell Hays, Kim Reinking, Josie Valdez, Connie Washino, and Barb Weston.

My friends have been a great help during this time, so thanks to the 512 F St. gang: Steve and Jennifer Nicholson, Todd Millick, and Ross Miller. Steve, thanks too for a few years of scintillating weekly conversations over good food at the Border! Thanks to Blayney Breckenridge and Sean Davis for your good friendship and fine evenings on L street. Whenever I've needed a break from Davis, it's been great to hook up with my buddies in Citrus Heights: Dave and Eileen, and Steve. Thanks for good tennis and good conversation.

I have always enjoyed the good banter in the security lab! Special thanks to the people I've known best. My personal SecLab Hall of Fame (Student Wing) would include these legendary names: Jeremy Frank, Steven Samorodin, Jason Schatz, Dave Klotz, Chris Wee, David O'Brien, Jeff Rowe, Marcus Tylutki, Steven Templeton, Dustin Lee, Scott Miller, Kirk Bradley, James Pace, Todd Heberlein, Tye Stallard, Dean Sniegowski, Ricardo Anguiano, Steven Cheung, and Christina Chung.

The first inductees to the Professor Wing of my SecLab Hall of Fame would be Karl Levitt and Matt Bishop. Thanks to both of you for all your support, including help on my qualifying exam and dissertation, and important financial help as well. Thanks to

Felix Wu, too, for being a friendly and helpful colleague.

Here I'd like to thank the group of people who have had the most influence in my life during my graduate school years. Thanks to Janicse Hardesty for being my partner through many of those years. Thanks to Biswanath Mukherjee and Ron Olsson for your consistent faith in me, and for inspiring me with your good characters. Thanks to Dave for your powerful good spirit and everything that flows from it, including generosity, humor, creativity, and friendship. It is good to be your brother. Finally, thanks to Mom and Dad for the strongest love that two sons could ever hope for.

Chapter 1

Introduction

As computers have become increasingly important components in society's infrastructure, interest in computer security has become more urgent. Over the last several years, a variety of approaches to computer security have evolved. We have used three such approaches in our research. The first approach is to protect computer systems with filtering mechanisms. This approach is quite natural for protecting against network threats. The second approach is to test systems extensively for vulnerabilities, and it is applicable to both computers and security systems that protect computers. The third approach is to add detection mechanisms to systems to recognize potential security breaches.

We used the filtering approach in two projects. We created a design and prototype for a filtering device called Safe Modem that protects computers from Internet threats. We also developed a filtering strategy to protect network server programs against message-flooding attacks. Testing was the focus in two further projects. We developed a methodology for testing intrusion detection systems. We also developed a methodology for testing network programs, based on a standard testing technique called partition testing. Another project emphasized detection: We contributed to developing a protocol called WATCHERS that detects a specific potential security problem in routers. In the remainder of this chapter, we introduce each of these projects.

1.1 Overview: Safe Modem

We designed Safe Modem to protect a single computer against several Internet threats, similar to the way that a firewall protects a network of computers against the same. Safe Modem is intended for home or small-office computers.

Safe Modem will intercept and examine each network packet sent or received by the protected computer. It will forward acceptable packets and discard packets that might cause security problems (or send replacement packets instead). Safe Modem will be capable of analyzing several different types of data for threats, including e-mail messages.

Safe Modem is a promising security device for several reasons. First, it combines several functions (e.g., e-mail scanning and firewall-like filtering) into one device. Second, the modular design of Safe Modem will allow new analysis components to be added to the system easily. Third, the Safe Modem design is adaptable, so Safe Modem can be implemented in several different ways. Finally, Safe Modem's design may lend itself to hardware acceleration, especially if Safe Modem is implemented as a separate device.

1.2 Overview: Testing Intrusion Detection Systems

Our first testing project involved testing intrusion detection systems (IDSs), which recognize activities in computer systems that might affect security. IDSs detect such activities by analyzing information such as operating system audit records and network traffic summaries.

As more and more IDSs are developed and organizations depend on them more for security, techniques for evaluating IDSs are becoming increasingly important. However, evaluating an IDS is a challenging task for at least two reasons. First, it is difficult to select a set of suspicious activities that the IDS should detect. (This set would be used to develop test cases for the evaluation.) Second, an IDS can be affected by various conditions in the computer system such as a high load of computing activity. This complicates the task of thoroughly testing the IDS.

We developed a methodology for testing IDSs that addresses these difficulties. The methodology consists of general software-testing techniques, which we have adapted for the specific purpose of testing IDSs. We first identified a set of desirable characteristics for an IDS such as the ability to detect a broad range of known intrusions. Then, we developed strategies for selecting test cases and detailed testing procedures. We constructed a software system (based on the Unix tool *expect*) for developing user-simulation scripts, which are run during the test procedures. Our enhancements to *expect* include mechanisms for concurrent scripts and a record-and-replay feature. Finally, we used the methodology to test an IDS called Network Security Monitor and we discovered a potential weakness.

1.3 Overview: Contributions to WATCHERS project

In another project related to detection, we helped to develop a protocol called WATCHERS which detects routers that drop or misroute packets. Such behavior might indicate a configuration problem or that an attacker has subverted a router and is now trying to disrupt communication.

The essence of WATCHERS is that each router monitors its neighbors for "conservation of flow": all data bytes sent to a router and not destined for that router are expected to exit the router. Each router also monitors its neighbors for signs of misrouting. When a misbehaving router is discovered, its neighbors stop communicating with it, and so it is logically removed from the network.

WATCHERS has some advantages over other network monitoring techniques that could be used for the same purpose. First, its robust communication strategy prevents attackers from interfering with WATCHERS itself. Second, WATCHERS can detect routers that misbehave only intermittently and routers that cooperate in an attempt to hide their malicious behaviors. Third, while some detection mechanisms are only able to report that a problem exists somewhere in the network, WATCHERS can often isolate the router or routers that are causing the problem. Fourth, we have proved that WATCHERS never mis-diagnoses a good router as bad in ideal conditions.

Our specific contributions to the WATCHERS project are as follows. We performed the complexity analysis of WATCHERS' memory and communication requirements and running time. We conceived of several different strategies that attackers might use to try to escape detection and demonstrated in detail that WATCHERS will foil most of them, but a few will succeed. We also helped to develop the proof mentioned earlier.

1.4 Overview: Message-Flooding Defense

In another filtering project, we developed a defense against message-flooding attacks against server programs on networks. In such attacks, the attacker inundates the server with service-request messages so that the server can no longer respond effectively to legitimate clients. These attacks can have severe consequences if the service is critical.

Our defense is designed for environments in which a group of clients is authorized to receive service but unauthorized clients might request service as well. The essence of the defense is that the server regularly inspects its queue of incoming messages and discards messages from unauthorized clients. The defense depends on a strong message-authentication method (for identifying unauthorized clients).

We proved that a server using the defense in ideal conditions can maintain good service to legitimate clients no matter how intense the flooding attack is. To evaluate the defense in more realistic conditions, we developed a software simulation. We found that a server using the strategy performed significantly better during a flooding attack than a server not using it. We also attempted to prove the strategy's worth analytically. We developed a preliminary mathematical model for a server using the strategy. However, we must overcome some challenging mathematical problems before we can complete our analysis of the model.

1.5 Overview: Testing Network Programs

In our second testing project, we developed a methodology for testing network programs. This is important because many security problems are caused by software problems in network programs. Our methodology is an example of a standard testing approach called "partition testing" in which the set of all possible inputs to a program is partitioned into subsets and one or more test cases are selected from each subset. In our methodology, the partitioning is based on the grammar that specifies valid incoming messages to the network program. We used the methodology to test an instance of the Unix "finger daemon" (fingerd).

An important step in our approach is the development of a "preprocessor" that filters some invalid messages based on some simple checks. The purpose of the preprocessor is to reduce the size of the set of potential valid messages that the network program might have to handle, to make testing more practical. After testing, the network program should always run in conjunction with the preprocessor. We developed a specification for a simple fingerd preprocessor.

When we tested fingerd according to our methodology, we found a flaw that, depending on the configuration of the finger service, could lead to a denial-of-service attack. This first round of testing was limited to test-case messages that our preprocessor would allow to pass. We next did a second round of testing in which we removed this limitation. Our goal was to find a software problem to demonstrate that it is difficult to completely test a fingerd program without using our methodology (including the preprocessor). We were successful: we found one test-case message that caused fingerd to malfunction.

1.6 Organization of Dissertation

In the succeeding chapters we discuss each of these research efforts in full detail. Chapter 2 describes the Safe Modem project. Chapter 3 details our work on testing intrusion detection systems. Chapter 4 discusses our contributions to the WATCHERS project. Chapter 5 describes our development of a message-flooding defense. Chapter 6 discusses our methodology for testing network programs. Chapter 7 concludes the dissertation.

Chapter 2

Safe Modem

2.1 Introduction

The *firewall* is a well-established security tool for protecting a network of computers against Internet threats. We now introduce a design (and a software prototype) for firewall-like protection for a single computer. We refer to the (future) implementation of the design as Safe Modem (SM). SM is targeted for a home or small-office computer.

We designed SM to monitor TCP/IP packets, since the Internet is based on the TCP/IP protocol suite. SM will intercept and analyze each IP packet sent or received by the protected computer. SM will analyze both the header and the data portion of each packet. Based on its analysis, SM will forward the packet, discard it, or send one or more replacement packets instead. (For example, such packets might deliver a warning message about an attempted attack to the user.) SM will include several analysis functions. Our SM prototype includes an IP packet header analyzer and an e-mail analyzer for SMTP (Simple Mail Transfer Protocol) packets. We chose to focus on e-mail protection since, as explained later, e-mail-based attacks are so prominent. Also, we can easily adapt the design of the e-mail analyzer to create analyzers for other protocols such as FTP and HTTP. In general, one strength of SM will be an extensible architecture that will make it straightforward to add additional analysis components.

For background, Section 2.2 describes some Internet threats. Section 2.3 discusses the SM architecture. Section 2.4 discusses each of SM's projected security features. Section 2.5 describes our performance testing of the SM prototype. Section 2.6 surveys related security tools. Finally, Section 2.7 summarizes why SM is a promising security device for the future.

2.2 Internet Threats

This section discusses several types of threats to a computer connected to the Internet.

2.2.1 Malicious Code

Malicious code is a program or part of a program that is designed to cause harm or damage when it runs. Three of the most prominent types of malicious code are computer viruses, Trojan Horses, and worms.

A *computer virus* is "a sequence of code that is inserted into other executable code, so that when the regular program is run, the viral code is also executed. The viral code causes a copy of itself to be inserted in one or more other programs [28]." The viral code might also perform a harmful action. A recent example is the "Melissa" macro virus (released in March 1999), a virus that attaches itself to Microsoft Word documents as a macro. The macro runs when the the document is opened (if the user has enabled macros). The Melissa macro sends itself (along with its host document) as an e-mail attachment to several (up to fifty) addresses listed in the victim's address book. The macro also attempts to change Microsoft Word settings so that future documents created by the victim will contain the Melissa virus [12].

While propagating itself widely, the Melissa virus had three types of harmful impact. First, the virus modified configuration data on each infected computer. Since tens of thousands of computers were affected, the time required to restore that data was substantial. Second, since Melissa propagated itself by sending itself *and* its host document via e-mail, it released potentially sensitive data. Finally, many mail servers had to be shut down either to avoid the virus or because of the high mail volume caused by the virus [79].

A second type of malicious code is a *Trojan Horse*, which is a program that performs a useful or desirable function, but also performs something malicious (often without the victim realizing it). For example, in one case, a malicious program was disguised as a "shareware" e-mail program. The program was fully functional and available for download from a number of popular shareware Internet sites. However, the program contained some malicious code that privately sent the user's e-mail passwords to a remote Internet site [58].

A third type of malicious code is a *worm*, which "is a program that is designed to copy itself from one computer to another, leveraging some network medium: e-mail, TCP/IP, etc. [77]." A prominent example is the "Love Letter Worm" (released in May 2000). The worm is a script which is run by a system called "Windows Scripting Host" on computers with a Microsoft Windows operating system (OS). When the script runs, it attempts to send a copy of itself as an e-mail attachment to each address in the victim's address book. Also, the worm searches for files with particular extensions (e.g., "jpg") and replaces those files with copies of itself. (It is likely that the original files cannot be recovered.) Finally, the worm attempts to down-load and install a password-stealing Trojan Horse program. The name of the program ("Win-bugsfix.exe") suggests that it fixes bugs associated with the Windows OS. However, if the victim runs the program, it sends passwords to a remote e-mail account [78].

One account of the Love Letter worm [46] (issued the day after the worm was widely detected) reported the following:

> ICSA.net [an Internet security firm] estimates that the virus has already infected more than a million computers causing in excess of $100 million in damage. The final bill will exceed $1 billion from lost data, interrupted work, and the cost of fixing the damage, ICSA.net chief scientist Peter Tippett said in a statement.

2.2.2 Buffer Overflow Attacks

Another threat to a computer connected to the Internet is an attack against a network program, a program that communicates with other computers over a network. A common type of attack against such programs is the *buffer overflow attack*, which has been described as the dominant form of attack among all remote penetration attacks [23]. In this attack, the network program is waiting to receive data. The attacker sends data to the program, but the data string is longer than the network program expects it to be. If the program fails to check the length, then the data string will likely overflow a buffer in the program. Several results are possible. The overflow might cause the program to terminate.

In some attacks, though, the data string is crafted so that the overflow causes the computer to begin executing code which is embedded in the attacker's data string. Often that code is executed with the same privileges as the original network program. Thus, for a UNIX computer, in the worst case, the attacker's code will be executed with root privileges. (Reference [23] discusses buffer overflow attacks in great detail.)

One factor that amplifies this threat is that a computer connected to the Internet often runs several network programs at the same time, including some that are not needed and not monitored. For example, a recent CERT advisory [13] about attacks against DNS servers includes this statement:

> The DNS server software was running because it was installed by default (unknowingly in many cases) when the machines were configured. This software was not up to date with security patches and workarounds; and since the system administrators were not planning to have the machines operate as DNS servers, they did not ensure the software was up to date, or simply disable the DNS server software on the machine.

2.2.3 Abnormal Network Packets

Still another threat is an attack triggered by an abnormal network packet, a packet with characteristics that would never occur during normal operation. Figure 2.1 lists several such attacks, along with signatures (the key identifying characteristics) of the abnormal packets associated with the attacks. (A reference is also provided after each attack name.)

Attack Name	**Signature**
Land [14]	(source IP address = dest IP address) & (source port = dest port)
Pepsi [33]	(source port = echo port) & (dest port = char gen port)
Ping o' Death [19]	total # bytes (as specified in IP header) > max. acceptable value
Smurf [33]	(packet type = ICMP echo request) & (dest = a broadcast addr.)

Figure 2.1: Attack signatures for abnormal packet attacks.

The abnormal packet might cause a computer's operating system to "crash", so that the computer must be restarted. Alternatively, the packet might trigger a storm of network traffic, which renders the computer unable to perform its usual activities. The general result is denial of service to legitimate users of the computer. If the computer is a source of revenue (e.g., if it runs a web server involved in e-commerce), the down time will have financial consequences.

2.3 Safe Modem Architecture

2.3.1 Overview

The projected primary SM components (and the data path through SM) are depicted in Figure 2.2. In operation, SM will sit between the protected computer and the Internet. The SM software will contains four components. An input component will read data, separate the data stream into IP packets, and insert the packets onto a queue. The "preliminary analysis" (PA) component will de-queue a packet and inspects the IP and TCP (if applicable) headers. For TCP packets, the PA component will associate each packet with its corresponding TCP connection. The third component will be a collection of specialized analysis (SA) functions, which can be invoked by the PA component. For example, when the PA component finds a packet destined for port 25 (the SMTP server port) on the protected host, it will invoke the e-mail scanning component. The SA component may choose to put the packet onto another queue, so that it can inspect further packets before deciding what to do. Both the PA component and SA components will be able to invoke the fourth component, which will simply send an IP packet to the protected host or Internet.

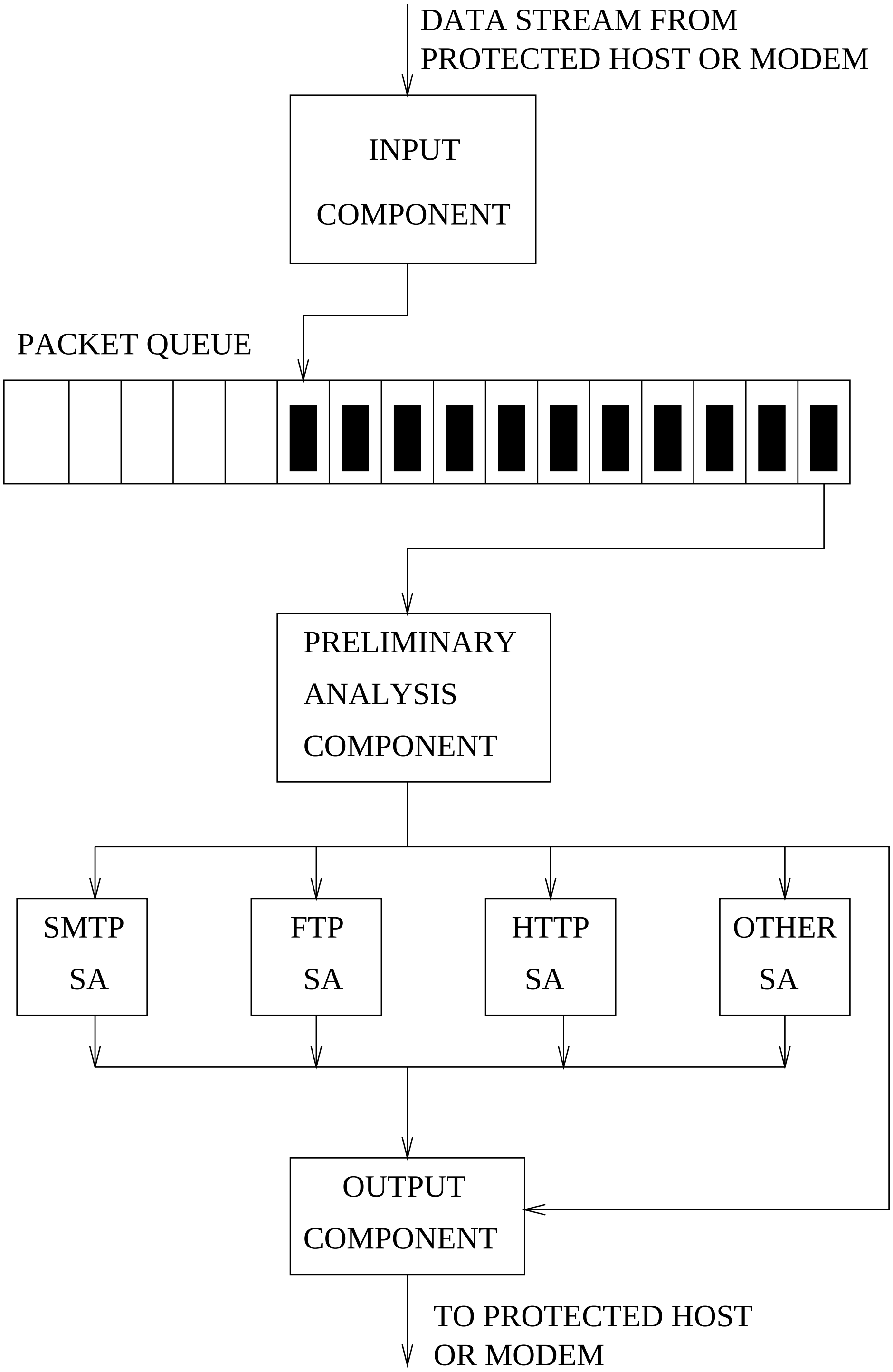

Figure 2.2: Primary Safe Modem components.

2.3.2 Prototype Architecture

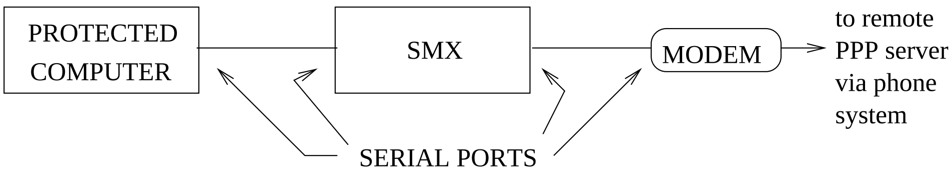

Figure 2.3: Prototype architecture.

Figure 2.3 shows the configuration in which our Safe Modem prototype (SMX) operates. SMX is implemented in software on a personal computer, and sits between the protected computer (PC) and an external modem. The PC connects to the Internet by establishing a PPP (Point-to-Point Protocol) connection over the modem to a remote PPP server. Then, it can communicate with remote Internet sites, sending and receiving IP packets encapsulated in PPP frames. SMX is nearly transparent to both the protected host and modem; they operate as if they were directly connected (except for some minor details in configuration files). SMX reads each PPP frame (from each side), extracts the IP packet, and analyzes it as described above. If the IP packet is acceptable, SMX forwards the PPP frame to the other side.

2.3.3 Implementation Issues

We visualize three different ways to implement SM. The first is simply a scaled-down version of the prototype. SM could be implemented as a circuit board with an inexpensive microprocessor, some main and long-term memory, and two serial ports. The circuit board could be an “expansion card” for a personal computer, or it could be packaged in a box, and used as an external device. The second option is to integrate SM technology into a modem. The modem could be a traditional phone-line modem, a cable modem, or a DSL (Digital Subscriber Line) modem. The third option is to implement SM in link-layer software to run on the protected host. For example, the SM technology could be incorporated into a PPP module.

2.4 Safe Modem Features

This section describes each of SM's projected security functions in detail.

2.4.1 Packet Filtering

A packet-filtering firewall decides to filter a packet based on values in the packet headers, especially the IP source and destination addresses, and the TCP source and destination port numbers [15].

SM will provide this same feature. For each packet, SM will extract some of the IP and TCP header field values and insert them into an array called a *packet vector*. SM will then compare the vector to a list of packet vectors (the *filtering rules*). If SM finds a match, it will discard the packet; otherwise, SM will forward the packet.

Some extensions to this feature are possible. First, each vector in the filtering rules could be associated with an "action phrase," indicating the action to be taken when that vector is matched. For example, the action phrases might include *discard*, *allow*, and *send warning message*.

As a more significant extension, an existing language for expressing filtering rules could be incorporated into SM's packet filter. For example, SnapL [51] is a language for specifying network access policies, which describe the packets and connections that are allowable on a network. The same language can be used (without changes) to specify filtering rules. Incorporating SnapL into SM in this way has at least two advantages. First, SnapL facilitates the task of writing filtering rules. Second, the SnapL implementation includes a "matching engine," which compares records of network traffic to policy rules. Adapting the SnapL matching engine to compare packet header values to filtering rules for SM would require less effort than creating an efficient matching engine from scratch.

2.4.2 E-Mail Filtering

SM will be able to filter e-mail messages sent to an SMTP server on the protected host. This feature works as follows for SMX. When a new SMTP connection begins, the SMTP analyzer in SMX starts to inspect each packet related to that connection. The analyzer is passive while the SMTP client and server exchange the first several packets, in which the two computers introduce themselves and the SMTP client identifies the sender and receiver of the mail message it is about to transfer. Eventually, the client sends the "DATA" command to the server, and the server responds by sending the numeric code 354, indicating that it is ready to receive the message. The client then sends one or more packets containing the message data. As SMX receives these packets, it does not forward them to the server. Instead, it copies the data from each packet to a buffer. After the last packet of the message arrives, SMX analyzes the message data as described below. If SMX determines that the message is acceptable, then SMX finally forwards the original packets to the server. If the message is not acceptable, then SMX discards the original packets, and constructs a new message. The new message contains the same header fields as the original, but the message body simply states that SMX had to drop the original message. SMX encapsulates the message inside an SMTP packet and sends it to the server. The message will be delivered to the intended recipient of the original message.

Message Analysis

SMX analyzes each e-mail message as follows. It first parses the message, locating the start and finish of each header field, the message body, and each attachment (if any) within the message buffer. SMX parses each attachment similarly; it locates the start and finish of each header field inside the attachment, and the attachment body. Thus, after parsing, SMX can analyze the data in any part of the message. Currently, SMX performs string matching on the message body and the attachment bodies. It checks for the presence of any of a set of strings using the Aho-Corasick string-matching algorithm [30]. SMX also decodes attachments that are Base-64 encoded before attempting the matching.

SMX should be extended to determine the type of data in the message body and in each attachment by inspecting the MIME (Multipurpose Internet Mail Extensions) content-type field. Then, SMX could invoke an appropriate scanning function. For example, SMX might invoke a Windows virus scanner for an attachment that is identified as a Windows executable file. Fortunately, it is straightforward to incorporate new analysis and translation functions into the SMTP analyzer.

2.4.3 Implementation Detail: TCP Spoofing

Before discussing SM's other data-scanning functions, we will describe a key implementation detail in SMX's e-mail scanner. When the scanner buffers packets and sends replacement messages, it can disrupt the transport layer (TCP) communication between the SMTP client and SMTP server. Here, we explain this problem in detail and then present our solution. The same solution can be used for other types of data scanning.

TCP Sequence Numbers and Acknowledgements

The TCP protocol includes several mechanisms to make communication more reliable. Two of them are sequence numbers and acknowledgements; they work as follows. Consider a TCP connection between a computer A and a computer B. Each data byte that A sends to B is (conceptually) assigned a sequence number, a 32-bit value. At the start of the connection, A selects an initial sequence number X (a 32-bit value). The first data byte sent to B has sequence number X+1 and the ensuing data bytes are numbered consecutively. The header of each TCP packet includes the sequence number of the first data byte in the packet. (If the packet contains no data, then the header sequence number is the sequence number of the next data byte that will be sent.)

When B receives a packet from A, B responds by sending an acknowledgement to B in one of its next packets. The acknowledgement information appears in the TCP header: B sets the ACK bit to 1 and sets the acknowledgment number (a 32-bit value) equal to the sequence number of the *next* data byte that B expects to receive from A. (Note: the sequence numbers and acknowledgments work in the same way for data transmitted from B to A.)

Safe Modem's TCP Spoofing Mechanism

The TCP sequence number and acknowledgment mechanisms create some problems for SMX's e-mail scanner. As we describe these problems and our solutions, we will make several references to Figures 2.4 and 2.5. The figures show most of the packets sent during an SMTP connection between a client and an SMTP server protected by SMX. (The packets associated with opening and closing the connection have been omitted to save space.) The three vertical lines represent the client, SMX, and the server. Each arrow represents a packet, and the arrow tails are numbered in the order in which they were sent. On top of each arrow, the range of sequence numbers corresponding to the data bytes sent in the packet is shown, and the acknowledgement number is shown. (Some packets carry no data, just an acknowledgment.) Below each arrow is a text string representing the packet data. (Only part of the data is shown in most cases due to space constraints.) The style of this diagram follows that in reference [76].

One problem is that, when SMX begins to receive the packets that contain the actual message from the client, SMX has to buffer those packets without forwarding them. However, the client is waiting for acknowledgements for the packets. If the acknowledgements do not arrive, the client soon ceases to send further packets and SMX may be stuck waiting for the last part of the message.

The solution to this problem is that SMX sends a "spoofed" acknowledgement to the client for each packet containing part of the message. That is, SMX constructs a TCP packet in such a way that it appears to have been created by the server and SMX sends this packet to the client. The TCP header in this spoofed packet contains the correct acknowledgement number. In Figure 2.4, packets 11, 13, and 15 are the packets carrying the e-mail message, and packets 12, 14, and 16 are the spoofed acknowledgements. If, after SMX forwards the packets to the server, the server sends *real* acknowledgements, the client

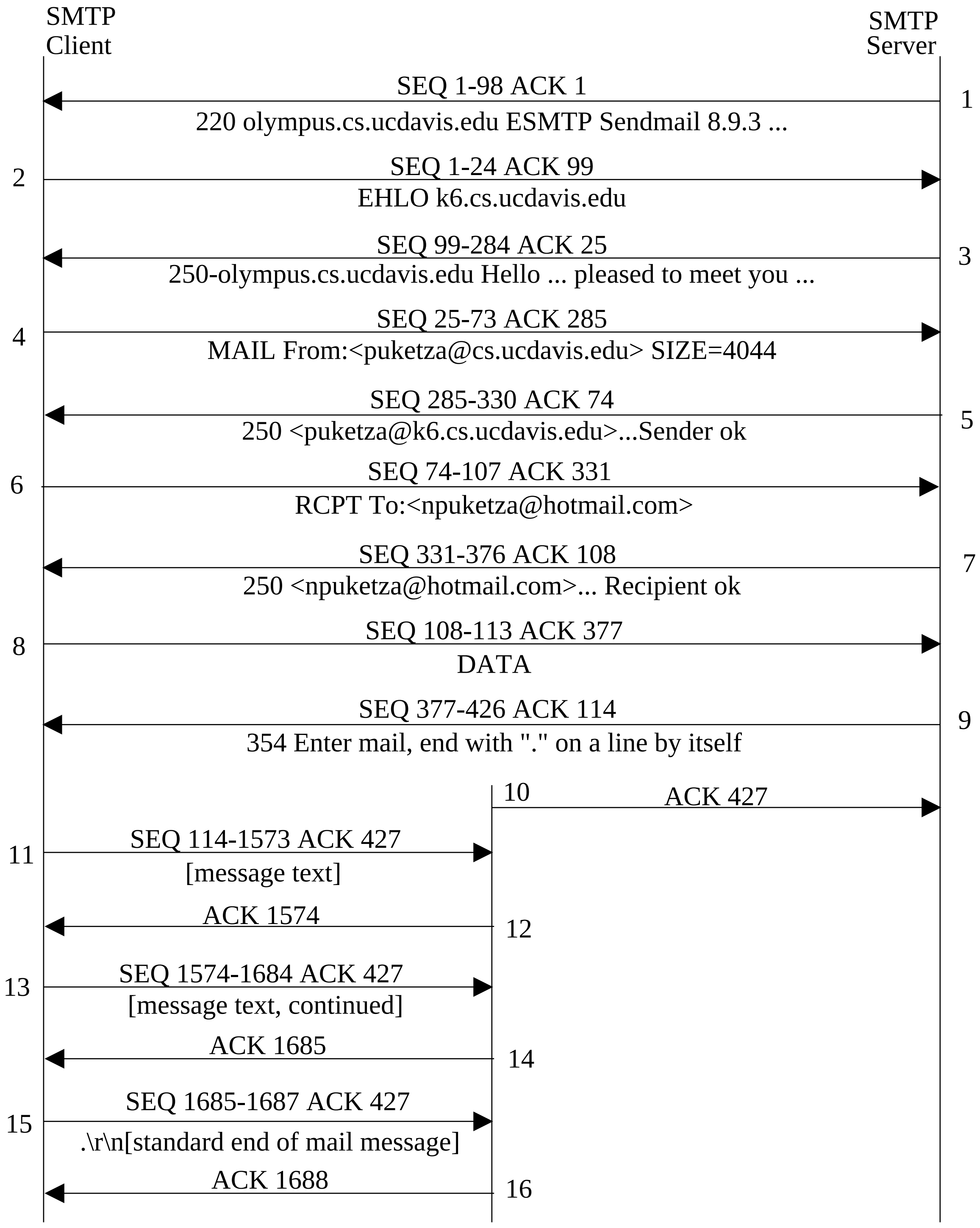

Figure 2.4: SMTP scanning.

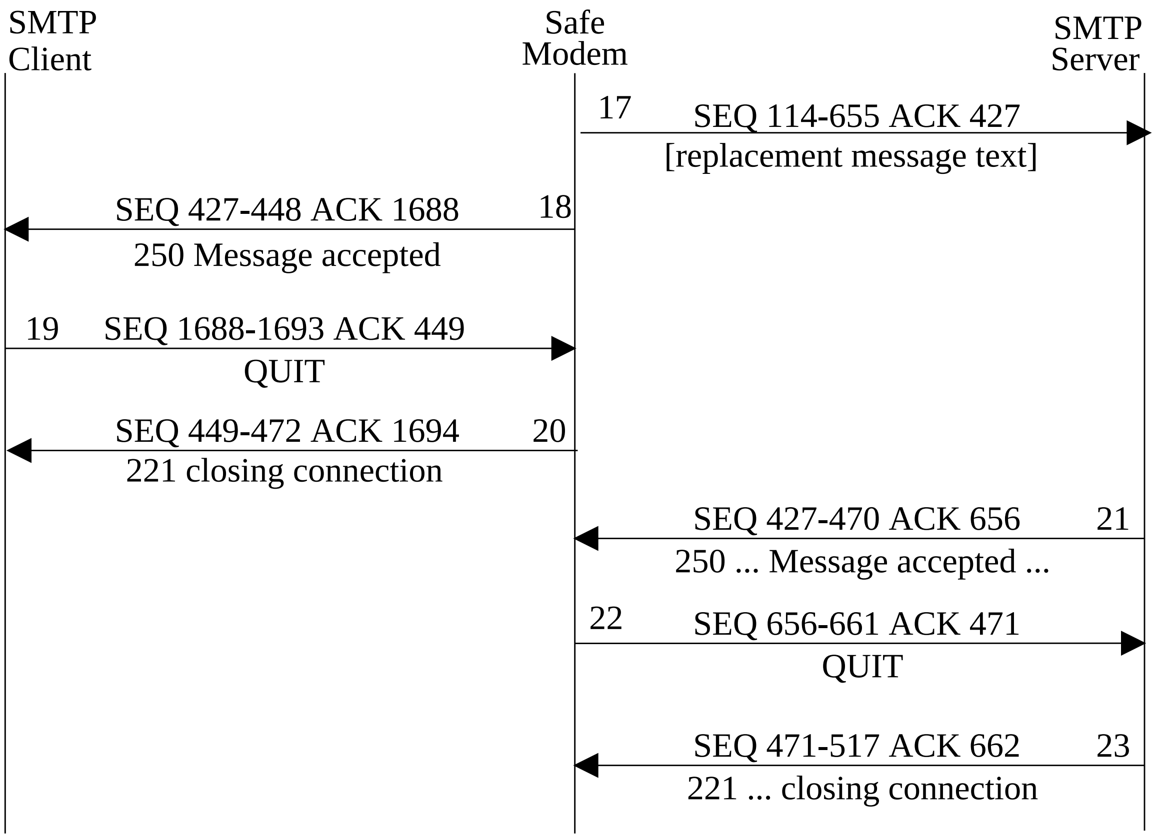

Figure 2.5: SMTP scanning (continued).

simply ignores them because they are duplicates (as per the TCP specification[37]).

Another problem occurs when SMX, after analyzing a message, decides to discard it and send a replacement warning message instead. The replacement message is not necessarily the same length as the original.[1] Thus, when the server receives the message, it will lose synchronization with the client with respect to sequence numbers and acknowledgement numbers.

SMX again uses spoofing to solve this problem. After it sends the replacement message, SMX sends a packet to the client indicating that the message was accepted. The message appears to the client as though it had come from the server. The acknowledgement number leads the client to believe that the entire original message was received and accepted. In Figure 2.5, packet 17 contains the replacement message and packet 18 contains the spoofed reply to the client.

SMX continues to masquerade as the server to the client for the duration of the connection. SMX intercepts each packet from the client, analyzes it, and sends back a packet with the appropriate reply, constructed as though the packet had come from the server. Packet 20 in Figure 2.5 is such a reply. Similarly, after sending the replacement message, SMX masquerades to the server as the client, as illustrated by packet 22.

2.4.4 General Approach to Scanning Data

The same approach that SMX uses to scan SMTP data can be adapted in a straightforward way for scanning data associated with other protocols.

[1] In the prototype, the body of the replacement message is the same in all cases. In future implementations, the replacement message may be customized based on the original message.

In general, the approach consists of these steps:

- monitor each command sent (in both directions);
- recognize a command that indicates that data will follow;
- buffer each packet carrying data;
- recognize the end-of-data indicator;
- write all the data to one buffer;
- parse the data if necessary;
- identify the type of data if possible;
- invoke an analyzer function on the data buffer; and
- based on the analyzer results, either forward the buffered packets, or discard the packets and perhaps take further actions such as sending replacement packets.

For example, this approach could be applied to scanning FTP data. A typical FTP connection proceeds as follows [76]. The client establishes a TCP connection to port 21 on the FTP server and sends FTP commands via this " control connection." Eventually, the client issues a command to request a data transfer to or from the server. A separate TCP connection (the "data connection") is used for the data transfer. Usually the client uses the control connection to specify which client-side TCP port the server should use for the data connection. The server then connects to that port, the data transfer takes place, and then the data connection is closed.

The standard server-side TCP port number used for the data connection is 20. Thus, a simple approach to scanning FTP data would be to examine all data in TCP connections involving port 20 on the remote host. However, some information passed through the control connection might be useful for SM when it scans the corresponding data in the data connection. For example, the file name extension (e.g., "html") might reveal the type of data being transferred. Also, perhaps future extensions to FTP will allow the server to use the control channel to directly specify the data type.

Thus, a more sophisticated scanning approach might be warranted. SM could monitor a control connection, waiting for a data transfer request. When such a request occurred, SM could then watch for the client's directions to the server about which port number to use for the transfer. After a connection to that port was established, SM could begin scanning data, using information gleaned from the control connection to scan more intelligently.

One challenge to our approach would be scanning a large file transferred via FTP (or as an e-mail attachment). Ideally, SM would scan an entire file before deciding whether or not to pass any of it to the protected host. However, SM might not have enough memory to buffer all of the data. In such cases, SM may have to scan the file in sections. When it finds a problem after it has already sent some data to the protected host, SM could send a warning message to the user, indicating that the file should not be trusted.

2.4.5 WWW Filtering

SM could be used to block access to particular pages on the World Wide Web (WWW). This feature would be desirable for parents who wish to prevent their children from viewing unsavory web pages, and for employers who wish to prevent employees from viewing web pages for entertainment during working hours.

This specific filtering feature is just one of several that SM could perform on WWW data according to the strategy outlined in Section 2.4.4. It is simple to implement. SM could monitor outgoing packets to port 80, the standard WWW server port. Each request for a web page starts with the "GET" command, an example of which appears below:

```
GET /images/csllogo.gif HTTP/1.0
```

```
Referer: http://seclab.cs.ucdavis.edu/
Connection: Keep-Alive
User-Agent: Mozilla/4.51 [en] (X11; I; FreeBSD 3.2-RELEASE i386)
Pragma: no-cache
Host: seclab.cs.ucdavis.edu
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
        image/png
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
```

After identifying a GET packet, SM could extract the hostname from the "Host" header field and compare it to a list of unacceptable hosts. If it discovered a match, SM could discard the GET packet or modify the packet so that it would travel to an acceptable site. For example, the modified request might result in the user receiving a web page that explains why they were denied access to the requested page.

2.4.6 Filtering Rule Updates

Several of the projected SM filtering features will require regular updates to their filtering rules. Thus, we intend for SM to have the ability to retrieve updated rule sets from a trusted site on the Internet. SM will use cryptographic techniques (e.g., digital signatures) to prevent an adversary from interfering with these transmissions.

2.4.7 IP Security

The IETF (Internet Engineering Task Force) has designed an IP Security Architecture (IPsec), which specifies the use of cryptography to protect IP packets from threats on the path from source to destination [40]. The essence of IPsec is that an IP packet can be encapsulated inside a second IP packet. The contents of the encapsulated packet can be encrypted and/or protected by a digital signature that is stored in the header of the enclosing packet.

SM could include a complete implementation of IPsec. The IETF IPsec RFC specifies that IPsec can be implemented by means of an "outboard crypto processor," where "outboard" means a device separate from the computer that performs the regular IP processing. Thus, even if SM is implemented as device external to the protected computer, it can still be used to implement IPsec. (See Section 2.3.3 for SM implementation options.)

A useful application of IPsec is virtual private networking (VPN). VPN is a technique by which a remote user can establish a secure connection to a computer at their home site, even though the connection goes through an untrusted public network (usually the Internet). For example, a traveling salesperson could use SM to establish a VPN connection from their computer to their home office through a phone line and the Internet.

2.5 Evaluation

The chief drawback of SM is that it adds a delay of time T to each packet P that flows through it. The value of T (which ranged from approximately 42 to 190 milliseconds in our filter tests described below) depends on these factors:

- direction of P;
- size of P (number of bytes);
- the particular SM functions to be performed on P;
- the SM configuration (e.g., number of filtering rules);
- data in P (e.g., the number of strings in the data that match SM patterns); and
- other packets in SM during P's passage.

We conducted some tests to investigate the impact of some of these factors on packet delay.

2.5.1 Filter Tests

We performed the filter tests to measure the packet delay caused by the firewall filter function (Section 2.4.1). This delay is important because the firewall filter function is usually applied to every packet during SM operation. We conducted the tests as follows. We used the UNIX "ping" program to send packets from the protected host A to a second computer B. Ping sends an "echo request" packet to the target computer; the target sends a reply in response. Then ping reports the "round trip time" (RTT), the time between sending the request and receiving the reply. A difference in RTT's for two separate tests indicates a difference in SM packets delays, since all other components of the RTT should be the same for all of the the filter tests. We also instrumented SM with code to measure the SM delay for each packet directly. For each ping request and reply, SM reports the delays for both packets. (We will refer to packets from A to B as *outgoing* and packets from B to A as *incoming*.)

For each trial, we instructed ping to send 100 requests, with one second between each request. (The one-second time between requests is enough that successive requests do not interfere with each other.) Ping reported the RTT for each request/reply pair and also reported the average RTT after receiving all the replies. For the first trial, we removed SM and connected the protected computer directly to the modem. For the second trial, we restored the normal configuration for SM operation, but we configured SM for zero firewall filtering rules. In each successive trial, we increased the number of filtering rules used by

SM.

The results are displayed in Figure 2.6. For each trial, there are four data points in the graph: average ping RTT, average delay for the incoming packets, average delay for the outgoing packets, and the sum of the two average packet delays. The horizontal line near the middle of the graph represents the average RTT for the first trial (with no SM).

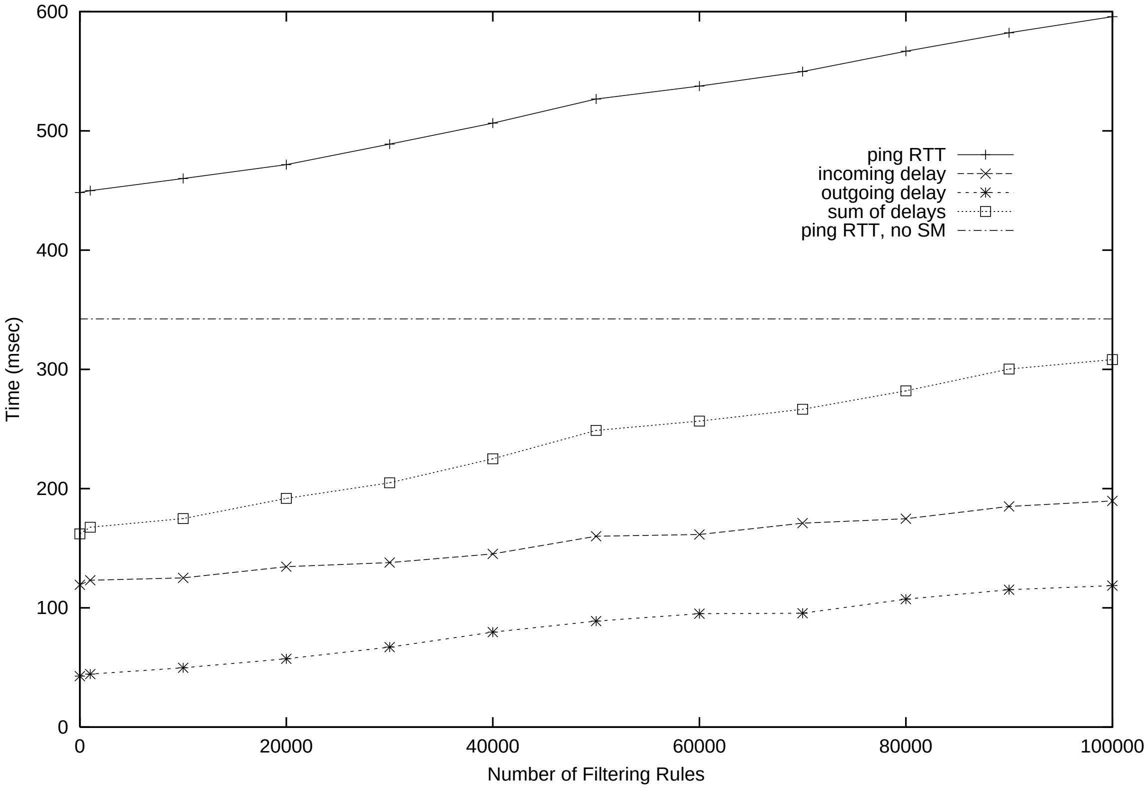

Figure 2.6: Filter test results.

The graph shows that packet delays increase linearly with the number of filtering rules, except for minor deviations. We expect that the deviations are due to factors in our test environment not under our control, such as:

- the behavior of the OS (operating system) on the SM machine as it reads and writes data;

- the behavior of the OS on machine B as it receives request messages and sends replies;
- network traffic unrelated to our experiments; and
- unrelated activity on the computers involved in our experiments.

Regarding the last two factors, machine B is connected to the internal network in our research lab. Traffic on that network caused by other users, especially traffic to machine B, could have affected B's response time. Furthermore, unrelated computing activity on machine B and on the remote PPP server could have affected the delay times as well.

Returning to the graph, we expected the linear relationship because the SM prototype checks every filtering rule sequentially for a match with each new packet.

The graph also shows that the ping-RTT curve matches the trajectory of the sum-of-delays curve. We expected this result because, between trials, we changed only the number of SM filtering rules, so any increase in RTT values should have been caused only by the same increase in packet delays. The graph indicates that the firewall-filtering function appears to scale well; for an increase of 100,000 filtering rules, the incoming packet delay increased just 70 milliseconds, and the outgoing packet delay increased 76 milliseconds. Future versions of SM will use a better algorithm to search for filtering rule matches, so scalability in this respect will only improve. This is important because we estimate that a list of filtering rules compiled now by a company that offers such lists as a product is likely to have tens of thousands of rules. Finally, the graph shows that the RTT values in all trials are not significantly worse than the average RTT value measured in the trial without SM. This result demonstrates that SM can be a practical security solution, that the SM user will not experience unreasonable delays.

2.5.2 Stress Test

When a packet arrives while SM is processing another packet, SM must insert the new packet onto a queue. When the arrival rate is low, SM can process packets as soon as they come in, and no queuing is necessary. If the arrival rate increases, it will reach a threshold value at which SM must first resort to queuing. We refer to this value as the *queuing point*. As the arrival rate increases further, the average length of the queue increases. Finally, the arrival rate will reach a second threshold value, above which SM is unable to keep up with the arriving workload, and the queue length grows until it reaches its maximum size. We refer to this value as the *flooding point*. Data loss will start to occur at the flooding point, because SM will be unable to store all of the arriving packets.

We conducted a stress test to investigate the behavior just described. We again used the ping program to send packets from the protected host A to a second computer B. For each trial, we instructed ping to send 100 requests packets, each carrying 500 bytes of data. In the first trial, we set the inter-departure time (IDT) between requests to 1.0 seconds, so SM had to process two packets (request and reply) per second. In each successive trial, we lowered the IDT, thereby increasing the packet arrival rate.

In Figure 2.7, the average RTT values for each trial are plotted. As a baseline, the horizontal line near the bottom of the graph represents the average RTT for the no-SM case. The graph demonstrates the behavior described above. A small jump in the graph suggests that the queuing point corresponds to an IDT between 0.5 and 0.55 seconds. Our queuing data (described below) confirms that SM first resorts to queuing when the IDT is 0.53 seconds. The graph also suggests that the flooding point corresponds to an IDT

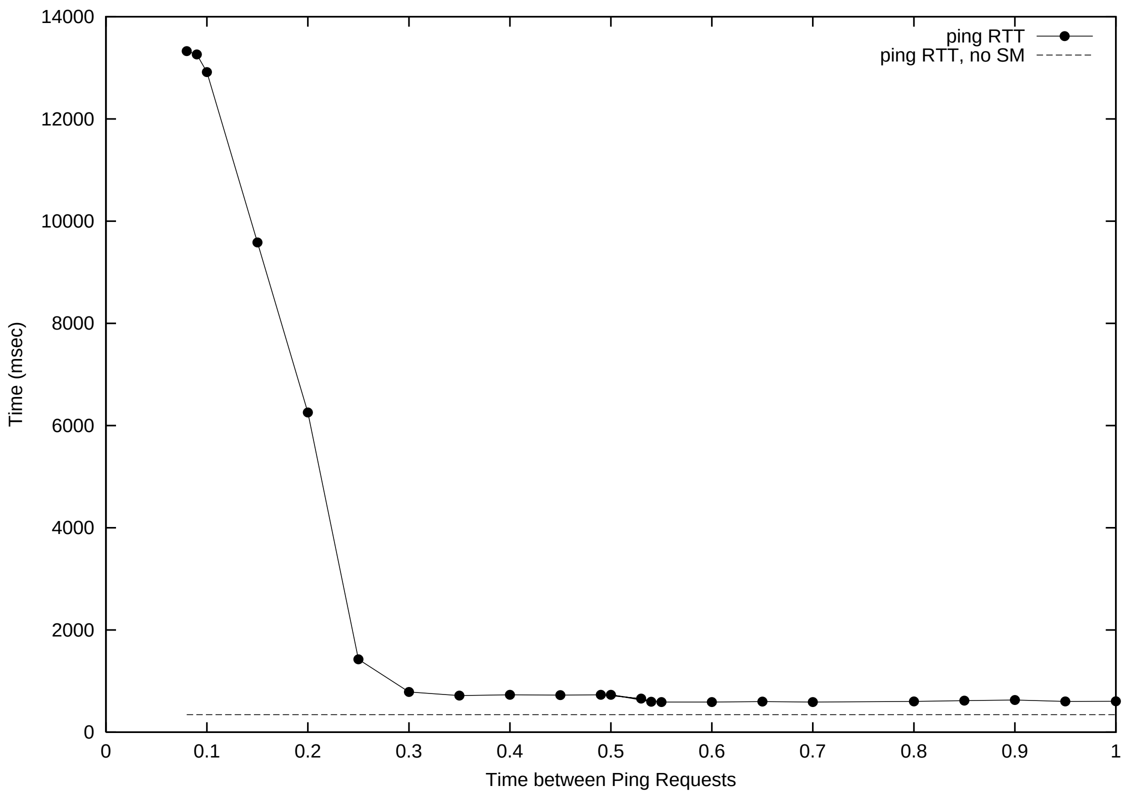

Figure 2.7: Stress test results.

between 0.25 and 0.2 seconds. Our queuing data confirms that the queue length grows rapidly when the IDT drops below 0.25 seconds.

The difference between the queuing point and the flooding point is worth noting. If packets arrived at SM in fixed intervals, and SM processed each packet in exactly the same amount of time, then the queuing point and the flooding point would be the same. The fact that they were not the same in our test illustrates that there was some variability in the time between packet arrivals at SM and in the SM processing times. For example, some packets must have arrived earlier than expected and in some cases SM must have had to insert those packets in the queue, but then later some packets must have arrived later than expected, allowing SM to "catch up."

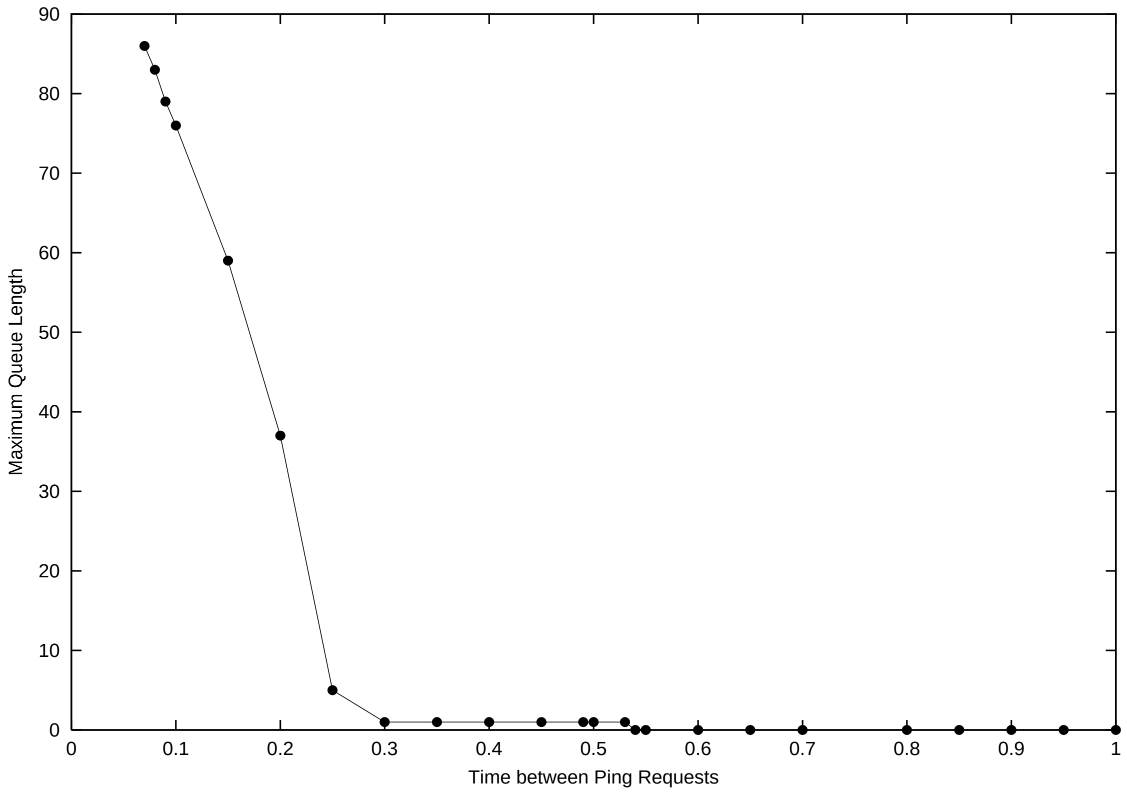

Figure 2.8: Stress test results.

The queuing data is shown in Figure 2.8. For each packet P, SM records the number of packets already in the queue when P arrives. In Figure 2.8, the largest of these values (the maximum queue length, or MQL) is plotted for each trial. The MQL is 0 until the IDT drops to 0.53 sec. When the IDT is 0.25, the MQL is 5, and when the IDT drops to 0.20, the MQL jumps to 37.

For IDT's less than 0.25 seconds, the data points on both graphs are somewhat misleading. The actual situation is worse than what the graphs show. For each trial in this range, the queue length increases steadily as ping sends more requests. Thus, had we sent more than 100 requests in each trial, the MQL and the average RTT would both have continued to increase (until, perhaps, SM started to malfunction).

This stress test was somewhat limited in scope. In our test, packets were sent (and then arrived at SM) at fixed intervals and all the requests and all the replies were nearly identical, so they required exactly the same processing time from SM. In future stress tests, the inter-departure time could become a random variable, and the characteristics of each packet (e.g., size in bytes) could be varied to make the required processing time a random variable also. Such tests might yield results that are more relevant to SM's normal operation.

Another evaluation technique would be to model SM as a queuing system. However, we expect that this would be complex task. For example, the SM prototype uses more than one queue: the OS of the SM computer has its own input and output buffers. Therefore, we believe that testing is the appropriate way to evaluate the SM prototype.

The delays we measured during testing (42 to 190 milliseconds) are tolerable for many network applications such as e-mail. However, such delays might be unacceptable for interactive applications in which users expect immediate responses to their keystrokes (e.g., telnet) and other input events, and for high-speed delivery of a stream of data (e.g., video). Thus, increasing SM's performance (e.g., by implementing a faster rule-checking algorithm) will continue to be an important part of SM development.

2.6 Related Security Devices

Several tools provide security functions similar to those planned for SM (though no tools we studied provided *all* of SM's projected features). Next we describe some of these tools and compare them to SM.

2.6.1 Ipfirewall

Ipfirewall (IPFW) [1] is packet-filtering-firewall software built in to the FreeBSD operating system. It has several attractive features. It supports filtering based on several different features in each packet (such as IP and TCP options), instead of just IP addresses and TCP port numbers. It supports several actions for matched packets beyond the basic "allow" and "deny," including forwarding to other IP addresses, diverting to other TCP ports, and sending TCP reset messages to the source when a packet is denied. IPFW supports logging when rules are matched. It provides a well-organized language for the user to specify rules. IPFW supports stateful filtering: a packet can be filtered based on the current state of the connection. (The chief use of this technique is to allow certain packets in only after the user's computer has initiated the connection to the outside.) An important characteristic of IPFW is that it is built into the FreeBSD kernel. This makes IPFW execute faster than a user-level filter, because the latter's performance is affected by costly context switches when it makes system calls to the operating system.

IPFW is significantly more advanced than the packet-filtering feature in the SM prototype (though it does not provide other functions that are in the SM design, including the e-mail scanner that is already implemented in SMX). However, we could enhance the

SM prototype to contain nearly all of the features of IPFW. That is, there is no fundamental incompatibility between the SM design and IPFW. SM could even be implemented so as to support the IPFW language for filtering rules.

2.6.2 Personal Firewall Software

Several existing software packages are advertised as “personal firewall” software and provide a suite of security functions to protect a personal computer. Norton Internet Security (NIS) 2000 was deemed best of these packages by one recent review [10]. It includes these features:

- Packet Filtering – NIS includes a traditional packet-filtering component, although the available documentation is not sufficient to compare it to IPFW. One review indicated that the feature is extremely flexible, though somewhat complex to use [69].
- Privacy Protection – The user can specify a list of text strings (such as credit card numbers, phone numbers, and addresses) that should not be permitted to be transmitted WWW servers on the Internet. NIS searches for these strings in outgoing packets destined for web servers, and blocks packets that contain them.
- Anti-Virus Software – NIS includes Norton Anti-Virus (NAV) software. In addition to recognizing viruses in programs and macros, NAV also scans e-mail attachments for viruses. The e-mail scanner is designed to scan messages as they travel from a POP (Post Office Protocol) server to a POP client on the user’s machine.
- WWW Cookie Blocking – When a user visits a WWW site, the web server often creates a “cookie”, which is some text containing information about that user. The

cookie is stored on the user's computer. When the user returns later to the same site, the server checks the cookie. For example, the cookie might contain a password that the user selected on their first visit to the site. Instead of asking the user to type the password, the server can simply check the cookie, and then allow the user access to the site [20].

However, cookies also cause privacy concerns. For example, consider the following business strategy used by a fictitious company we will call Tracker. (This strategy is used by a real company [21].) Tracker's clients are companies that sell products on the Internet. When a user visits the web site of a Tracker client for the first time, a Tracker cookie is stored on the user's computer. After that, whenever the user visits the web site of any Tracker client, the web server reads the Tracker cookie (which serves as the user ID). The web server sends the cookie to Tracker, which updates its list of web sites that the user has visited. Tracker also returns a summary of this information to the web server, so that it can choose which ads to display to that user.

NIS allows the user to selectively block cookie storage. For example, the user could block all Tracker cookies, but still allow cookies from a news-related site.

- Internet Filter – NIS blocks access to a list of potentially objectionable WWW sites.

NIS's most significant advantage over SM is the anti-virus feature. SM's string matching facility could be used to search for viruses in incoming files, but NAV is more sophisticated. However, several of the above features could be implemented in SM. In particular, SM's string matcher makes it straightforward to add the privacy protection feature. The WWW cookie-blocking feature could be incorporated into SM's scanning functions, according to the general strategy outlined in Section 2.4.4. SM's potential capability for Internet filtering is discussed in Section 2.4.5. Furthermore, we expect SM to have features not found in NIS, including IPsec, FTP data scanning, and more extensive scanning of WWW data.

2.7 Summary

This section summarizes the features that make SM a promising security device.

One useful characteristic of SM will be that it combines several functions into one device. This will simplify the configuration of security options for a user. A single user interface can control the configuration options for all of the functions. Also, when updates are needed, a single network session can deliver the updated code for all of the functions.

A second key SM attribute is that it will be extensible. The design is such that new data-scanning modules can be easily integrated into SM. Furthermore, the prototype TCP-data-scanning approach, used in the SMX e-mail scanner, should be applicable to other protocols as well (e.g., POP, SMTP, FTP, and HTTP).

A third attribute is that the SM technology will be adaptable. Several implementations are possible:

- Like the SM prototype, SM could be implemented as a separate device, apart from the protected PC and the modem;
- SM could be integrated into link-layer software (e.g., PPP) on the protected host; or
- SM could be integrated into a communications device such as a v.90 modem or DSL/cable modem. (In fact, Safe Modem derives its name from this idea.)

Finally, SM's modular design may lend itself to hardware acceleration, especially if SM is implemented as a separate device. For example, a special chip could be developed to be a string-matching engine. In the case of e-mail scanning, the SM main processor could parse the message, then stream the data in each attachment to the string-matching chip, and finally read the results from the chip.

Chapter 3

TIDS

3.1 Introduction

An intrusion detection system (IDS) is a system that attempts to identify unauthorized uses, misuses, or abuses of computer systems by either authorized users or external perpetrators [53]. (For convenience, we refer to all such actions as *intrusions*.) Some IDSs monitor a single computer, while others monitor a group of computers connected by a network. IDSs detect intrusions by analyzing information about user activity from sources such as audit records and network traffic summaries. IDSs have been developed and used at several institutions.

Some current example IDSs are SRI International's *Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD)* system [57], *Network Flight Recorder (NFR)* by Network Flight Recorder Inc. [64], *Snort* by Martin Roesch [65], UC Santa Barbara's *NetSTAT* [80], and UC Davis' *Graph-Based Intrusion Detection System (GrIDS)* [74]. Three other IDSs that will be discussed further in this chapter are SRI International's

Intrusion Detection Expert System (IDES) [50, 49], and UC Davis' *Distributed Intrusion Detection System (DIDS)* [72] and *Network Security Monitor (NSM)* [31].

As more and more organizations depend on IDSs as key components in their computer security systems, techniques for evaluating IDSs are becoming more important. IDS users need to know how effective their IDSs are, so that they can decide how much they can *rely* on their IDSs, and how much they must rely on other security mechanisms. However, evaluating an IDS is a difficult task for a couple of reasons. First, it can be difficult or impossible to identify all possible intrusions that might occur at the site where a particular IDS is employed. The number of known intrusion techniques is quite large (e.g., see [56]). Also, the site may not have access to information about all of the intrusions that have been detected in the past at other locations. Furthermore, intruders can discover previously unknown vulnerabilities in a computer system, and then use new intrusion techniques to exploit the vulnerabilities. A second reason why evaluating an IDS is difficult is that an IDS can be affected by various conditions in the computer system. For example, even if an IDS can ordinarily detect a particular intrusion, the IDS may fail to detect that same intrusion when the overall level of computing activity in the system is high. This complicates the task of thoroughly testing the IDS.

We have developed a methodology for testing IDSs which confronts these difficulties. As the basis for the methodology, we have identified a set of general IDS performance objectives, such as the ability to detect a broad range of known intrusions. The methodology is designed to measure the effectiveness of an IDS with respect to these objectives. It consists of strategies for selecting test cases, and a series of detailed testing procedures.

To develop the methodology, we borrowed techniques from the field of software testing and adapted them for the specific purpose of testing IDSs. We use the UNIX tool *expect* [47] as a software platform for creating user-simulation scripts for testing experiments. In addition, we enhanced *expect* with features for simulating more-sophisticated intrusions, and a feature that greatly facilitates script creation. Neither the methodology nor the software platform is specific to a particular IDS, so they can be used to test and compare several different IDSs.

Our work should be useful to IDS developers, who can use our methods and tools to supplement their own approaches to testing their respective IDSs. System administrators can use our work to check for weaknesses in the IDSs that they currently employ. An organization that plans to acquire an IDS can use our work to compare the relative strengths and weaknesses of several IDSs and choose the system that best fits their computing environment.

As background, Section 3.2 in this chapter presents some examples of intrusions, provides motivation for intrusion detection, and discusses specific approaches to intrusion detection. Section 3.3 describes a software platform for testing experiments, which consists of *expect* and our enhancements. Section 3.4 begins the discussion of our testing methodology by first identifying a set of important IDS performance objectives, and then discussing the selection of test cases, and some limitations to our approach. Section 3.5 describes our specific procedures for testing experiments in detail. Section 3.6 presents quantitative results from our own testing experiments which we conducted on NSM. Section 3.7 concludes the chapter.

3.2 Background

In this section, we present some information on intrusions and intrusion detection as background for our work.

3.2.1 Intrusions

Intrusions in computer systems are occurring at an increasingly alarming rate. Some sites report that they are the targets of hundreds of intrusion attempts per month [4]. Moreover, there are numerous different intrusion techniques used by intruders [56]. The following scenarios are examples of intrusions:

- An employee browses through his/her boss' employee reviews;
- A user exploits a flaw in a file server program to gain access to and then to corrupt another user's file;
- A user exploits a flaw in a system program to obtain *super-user* status;
- An intruder uses a script to "crack" the passwords of other users on a computer;
- An intruder installs a "snooping program" on a computer to inspect network traffic [29], which often contains user passwords and other sensitive data; and
- An intruder modifies router tables in a network to prevent the delivery of messages to a particular computer.

The reader can easily infer some of the consequences of intrusions from the preceding list. Some additional consequences include loss or alteration of data, loss of money when financial records are altered by intruders, denial of service to legitimate users, loss of trust in the computer/network system, and loss of public confidence in the organization that is the victim of an intrusion [29].

3.2.2 Concurrent Intrusions

Intrusions can take one of two forms: sequential or concurrent. In a *sequential intrusion*, a single person issues a single sequence of commands from a single terminal or workstation window. In a *concurrent intrusion*, one or more intruders issue sequences of commands from several terminals, computers, or windows. The command sequences work cooperatively to carry out an attack. For example, an intruder can open multiple windows on a workstation and connect to a target computer from each window. Multiple intruders can try to conceal an intrusion attempt by distributing the suspicious behavior amongst themselves. Since window interfaces are ubiquitous, such concurrent intrusions are very likely to occur, so an IDS should be able to detect them.

3.2.3 Motivation for Intrusion Detection

Intrusion detection is a practical approach to enhancing the security of a computer system. Traditional approaches to computer security, such as efforts to develop strong access-control mechanisms, have focused on completely *preventing* unauthorized computer activities. However, for many computer systems it is unlikely that all vulnerabilities can be eliminated, due to the complexity of computer system design, the possibility of configuration and administration errors, and the potential for authorized users to abuse their privileges [53]. Security problems in computer networks must also be considered. For instance, there are a number of security flaws inherent in the widely-used Transmission

Control Protocol/Internet Protocol (TCP/IP) suite, regardless of its particular implementation [3]. When a system's vulnerabilities are exploited, intrusion detection can provide a second level of defense. Also, the intrusion detection approach is attractive because it does not require the expensive replacement of existing computer systems. (Similar arguments for intrusion detection appear in reference [24].) Finally, there is published evidence that an IDS can improve the process of identifying real intrusions. One report [48] indicates that, by using an IDS, TRW was able to speed up its investigations of intrusions. A second report [53] indicates that researchers using an IDS were able to identify many intrusions and intrusion attempts which were not detected by the standard methods of the system administrators at the site.

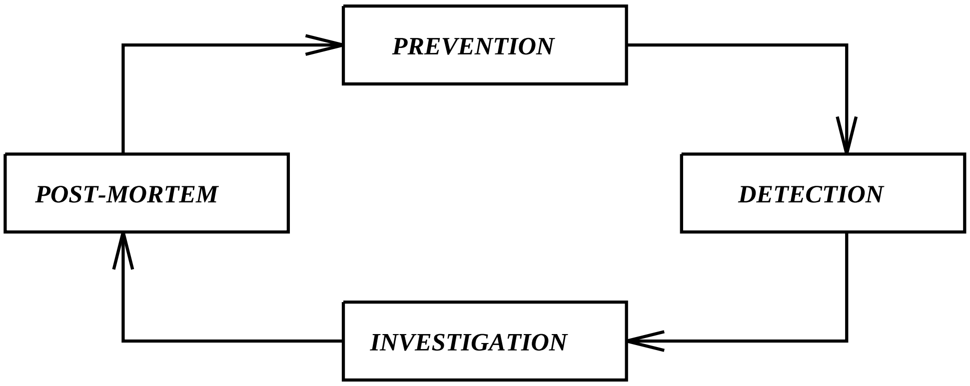

Figure 3.1: A computer system security management model.

In response to these difficulties in developing secure systems (which are discussed further in [53]), a new model of system security management [2] has emerged. The model is pictured in Figure 3.1. In this more-realistic approach to security, developing secure systems (the *prevention* component) is just one of four parts of the security system. The *detection* component identifies security breaches. The *investigation* component determines exactly what happened based on data from the detection component. This component may also include the gathering of further data in order to identify the security violator. Finally, the *post-mortem* component analyzes how to prevent similar intrusions in the future. In the past, most of the attention of computer security researchers has been focused on the *prevention* component. As IDSs prove to be useful, the *detection* component is beginning to receive more attention. Unfortunately, the other two components in Figure 3.1 have not yet received sufficient attention. In summary, the new model divides the field of computer security into four non-trivial and challenging but more manageable sub-problems; also, it encourages security researchers and practitioners to distribute their efforts over *each* of these important components of computer security.

3.2.4 Approaches to Intrusion Detection

The two major approaches to intrusion detection are called *anomaly detection* and *misuse detection*. The anomaly-detection approach is based on the premise that an attack on a computer system (or network) will be noticeably different from normal system (or network) activity, and an intruder (possibly masquerading as a legitimate user) will exhibit a pattern of behavior different from the normal user [24]. The IDS attempts to characterize each user's normal behavior, often by maintaining statistical profiles of each user's activities [49, 38]. Each profile includes information about the user's computing behavior such as normal login time, average duration of login session, CPU usage, disk usage, favorite editor, and so forth. The IDS can then use the profiles to monitor current user activity and compare it with past user activity. Whenever the difference between current activity and past activity falls outside some predefined bounds (threshold values for each item in the profile), the activity is considered to be anomalous, and hence suspicious. (The interested reader is referred to [38] for a thorough discussion of the IDES anomaly detector.)

A limitation of anomaly detection is that it depends on consistency in the behavior of users. In some environments, legitimate users may frequently change their behavior. For those users, it is difficult to create profiles that are flexible enough to tolerate legitimate variations in behavior, yet sensitive enough to detect intrusions.

In the misuse-detection approach, the IDS watches for indications of "specific, precisely-representable techniques of computer system abuse" [43]. In general, a misuse detection system consists of two key components. The first is a database of intrusion *signatures*, which are encapsulations of the identifying characteristics of specific intrusion techniques. The second is a pattern-matching mechanism which searches for the intrusion signatures in the records of user activities (e.g., audit trails).

Anomaly detection provides a means to detect masqueraders or legitimate users abusing their privileges without requiring knowledge of security flaws in the target system [24]. Misuse detection, on the other hand, can identify intrusive behavior, even when the behavior appears to conform with established patterns of use (i.e., the established user profiles). For this reason, several IDSs employ both an anomaly detection component and a misuse detection component.

Often, the main source of information about user activity for an IDS is the set of audit records from the computer system. However, relying on audit records *alone* can be problematic [53]. First, audit records may not arrive in a timely fashion. Some IDSs use a separate computer to perform the analysis of audit records (e.g., Haystack [70]), and it may take substantial time to transfer the audit information from the monitored computer to the computer which performs the analysis. Second, the audit system itself may be vulnerable. Intruders have been known to be able to turn off the audit system or to modify the audit records to hide their intrusions. Finally, the audit records may not contain enough information to detect certain intrusions. For example, in the so-called *doorknob attack* [72], an intruder tries to guess passwords of accounts on several computers in a network. To avoid arousing suspicion, the intruder attempts only a few guesses on each individual computer. This intrusion is not likely to be detected by analysis of the audit records of any single computer in the network. We use the term *network intrusion* to refer to such an intrusion that involves more than one computer (or other component) in a network.

This example illustrates that an effective IDS should also collect and analyze information from the network itself. For example, NSM monitors network traffic on a Local Area Network (LAN), so NSM can detect security-related network events such as the transfer of a password file across a network. DIDS collects and analyzes information from both the group of monitored computers and the network. Thus, DIDS is capable of recognizing network intrusions such as the aforementioned doorknob attack. A number of other scenarios in which the aggregation of information from a network of computers is necessary to detect intrusions is described in reference [72].

To conclude this section, we provide a detailed description of an IDS as an example. In the case of DIDS (see Figure 3.2), each monitored computer runs a Host Monitor

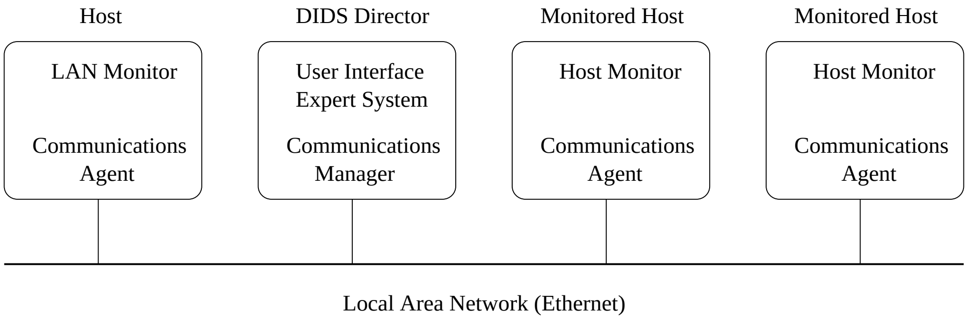

Figure 3.2: The Distributed Intrusion Detection System (DIDS).

program, which filters and analyzes the audit records associated with activity on that particular computer. The Communications Agent program on each monitored computer sends information to a central computer designated as the DIDS Director. Programs that run on the DIDS Director include: (1) an expert system to analyze the aggregated information from the monitored sources; (2) a communications manager to control the information flow for the entire system; and (3) a user-friendly interface for the SSO (System Security Officer). An additional computer runs the LAN Monitor program, which, like NSM, monitors network traffic in the LAN. Like the monitored computers, the LAN Monitor communicates with the DIDS Director via a Communications Agent program. With this architecture, intrusions on individual computers in the LAN can be detected by the Host Monitor programs, while network intrusions such as the doorknob attack can be detected by the DIDS Director, using information from the LAN Monitor and each of the monitored computers.

3.3 Software Platform

Our testing methodology is based on simulating computer users—both intruders as well as normal users—while the IDS is running. We use scripts to simulate users, and the scripts are interpreted and run by our software platform. The software platform is based on the *Expect* package [47] and the Tcl Distributed Programming (*Tcl-DP*) package [71]. *Expect* and *Tcl-DP* are both based on another UNIX package called *Tcl* (Tool command language) [60]. The *Tcl* package provides an interpreter for a programming language that includes variables, procedures, control constructs such as "if" and "for" statements, arithmetic expressions, lists, strings, and other features. Recent versions of *Tcl* also provide string-matching commands for matching strings against regular expressions. The syntax of *Tcl* shares similarities with the syntaxes of the UNIX shells and Lisp. *Tcl* is implemented as a C library package.

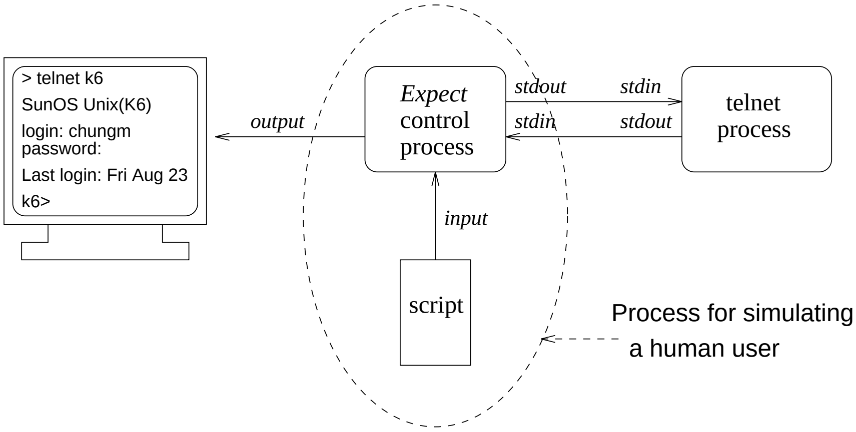

Figure 3.3: Simulation of a human user's activity.

The *expect* package also provides a programming language interpreter. The core of the *expect* interpreter is the *Tcl* interpreter, but *expect* extends the *Tcl* command set to include several commands for controlling interactive programs. The command "spawn" creates an interactive process (such as *telnet*). The command "expect" waits to receive a specified string pattern (such as "login: ") from the process. The command "send" sends a string to the process. Thus, a script containing several "expect/send" sequences and general programming constructs (such as variables, procedures, conditionals, and loops) can control an interactive session and thereby simulate a human computer user. Figure 3.3 illustrates the operation of *expect*. The interactive session in Figure 3.3 is the telnet session. *Expect* connects the standard input (*stdin*) and the standard output (*stdout*) of the telnet process to the control process. The control process issues a sequence of commands from a script to the telnet process, and the output of these commands is displayed on the terminal that invokes the execution of the script. Thus, the simulation produces the same output as a human user that issues the sequence of commands interactively.

The following is a simple *expect* script that controls a brief *rlogin* session:

```
#Spawn an rlogin process.
spawn rlogin ComputerName -l UserName
#Expect the password prompt, then send the password.
expect {"Password:" send "ActualPassword \r"}
#Expect the shell prompt, then send commands.
#The shell prompt is specified in a regular expression.
expect {-re ".*%|.*>|.*#" send "whoami \r"}
expect {-re ".*%|.*>|.*#" send "ls \r" }
expect {-re ".*%|.*>|.*#" send "logout \r" }
```

The *expect* package by itself provides the capability to create a script to simulate a computer user. *Tcl-DP* provides a suite of commands for creating client-server systems. We have augmented these packages with some additional commands that provide the capability to create *concurrent* scripts, complete with mechanisms for synchronization and communication among different scripts [82, 17, 63]. These extensions to *expect* provide users with the ability to simulate concurrent intrusions, which were described in Section 3.2.2. We have also developed a record-and-replay feature to facilitate script creation.

3.3.1 Support for Concurrency

It is often necessary or desirable to repeat an IDS test. For example, we can repeat a test to determine why (or why not) the IDS failed the test. In the repeated test, the sequence of events in the execution of the test scripts should be the same as in the original test. To meet this condition, the testers must use synchronization techniques for concurrent script sets. Otherwise, there is no guarantee that the *overall* sequence of events in the execution of a concurrent script set will stay the same from test to test, even if the order of events within each individual process is fixed. Randomness in system operations (e.g., variations in the duration of disk I/O events) can cause this non-determinism in the execution sequence of a concurrent script set, resulting in a test that is not *reproducible.*

Synchronization is also often necessary just to make a concurrent script set work correctly. Consider a concurrent password-cracking intrusion, in which three intruders collaborate in an attempt to crack passwords on a target machine. Figure 3.4 lists the specific tasks for each intruder during the intrusion. One intruder is responsible for copying the cracker program from a remote machine and for cleaning up after the intrusion is completed. The second intruder is responsible for compiling and running the cracker program when the program is available on the target machine. The third intruder is responsible for checking the output generated by the cracker program and for logging in to the target system if a password is cracked. In this concurrent intrusion, several commands issued by the intruders have to be synchronized (as indicated by the arrows in Figure 3.4). For example, the uppermost arrow indicates that Intruder 1 must create the "attack" directory before Intruder 2 can change to that directory.

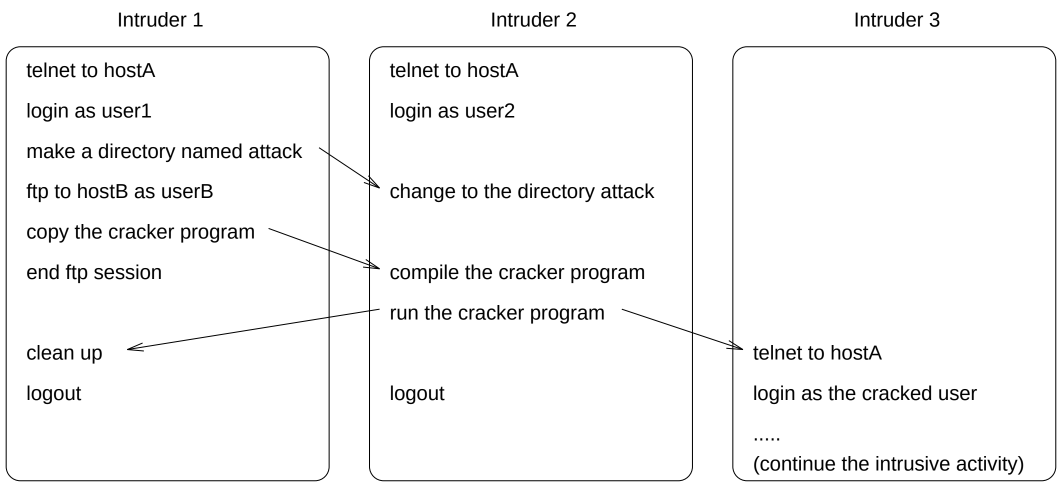

Figure 3.4: Concurrent password-cracking intrusion.

To support scripts with such synchronization requirements, and to support "reproducible testing" [9, 45], our software platform includes a synchronization mechanism. The mechanism provides a means for the programmer to establish a fixed order of execution for key events, even if the events are associated with different scripts. Furthermore, the mechanism is flexible, so that the programmer can easily modify the order of events. This flexibility should be useful during testing experiments. The testers can run a concurrent script set several times, each time with a different synchronization specification, and they can then observe if the changes in synchronization affected the IDS's reaction to the commands in the scripts, or the IDS's ability to detect intrusions.

Our platform also provides concurrent processes the ability to communicate with each other while executing a concurrent script set. This feature helps to create better simulations of groups of intruders working together.

3.3.2 Record and Replay

An additional feature that we have incorporated into our software platform is a "record-and-replay" capability. A user can type in a sequence of commands manually, and use the record feature to record that sequence. The "recording" can then be replayed at will, just like other scripts. Also, it is possible to set "synchronization points" in the recorded session, so that it can be replayed in synchronization with other recorded sessions. With the record-and-replay feature, a user can create an intrusion script quickly and easily without knowledge of *Tcl* and *Expect*. In particular, this feature should be most useful to system managers when they wish to create *site-specific* scripts to simulate *site-specific* intrusion techniques, which are described in Section 3.4.2.

3.4 Testing Issues

Now, we begin the discussion of testing methodology by examining some key testing issues.

3.4.1 Performance Objectives for an IDS

The first step in our IDS testing methodology is to identify a set of IDS performance objectives. We have identified the following objectives (which are similar to the key design goals for developing an IDS cited in [42]):

- **Broad Detection Range:** for each intrusion in a broad range of known intrusions, the IDS should be able to distinguish the intrusion from normal behavior;
- **Economy in Resource Usage:** the IDS should function without using too much system resources such as main memory, CPU time, and disk space; and
- **Resilience to Stress:** the IDS should still function correctly under stressful conditions in the system, such as a very high level of computing activity.

An IDS should meet the first objective or else many intrusions will escape detection. The second objective is required because, if an IDS consumes too much resources, then using the IDS may be impossible in some environments, and impractical in others. Finally, the third objective is necessary for two reasons: (1) stressful conditions may often occur in a typical computing environment; and (2) an intruder might attempt to thwart the IDS by creating stressful conditions in the computing environment before engaging in the intrusive activity. For example, an intruder might try to interfere with the IDS by creating a heavy load on the IDS host computer. Thus, we claim that it is *necessary* (though perhaps not *sufficient*) for an IDS to meet these three objectives in order to be effective in a wide range of computing environments. Our testing procedures are designed to measure the effectiveness of an IDS with respect to these objectives.

At different sites these objectives will have different relative values. A broad detection range may not be necessary if the IDS monitors a site that is protected from many attacks by other security mechanisms. For example, the site might use a firewall to block most incoming network traffic, and strict access control and authentication techniques to prevent abuse by insiders. Economy in resource usage may not be required at a site where security is a high priority and where computing resources exceed user needs. Finally, resilience to stress may be less important if controls in the computing environment (e.g., disk quotas and limits on the number of processes per user) prevent users from monopolizing resources. Thus, the most important objectives for a particular site should be identified by system administrators before an IDS is evaluated, and the IDS tests should be developed accordingly.

3.4.2 Test Case Selection

In our approach a *test case* is a simulated user session. While some of the tests require simulated "normal" sessions, most of the test cases are simulated intrusions. A key problem is to select *which* intrusions to simulate. The testers should first collect as much intrusion data as possible. For UNIX systems, [41] and [5] report that intrusion data can be obtained from various sources, such as CERT advisories, periodicals such as PHRACK and 2600, and the USENET [26], and also by analyzing the vulnerabilities detected by security tools such as COPS [25] and TIGER [67]. Next, assuming that the number of intrusions is too large to simulate all of them, the testers must partition the set of intrusions into classes, and then create a representative subset of intrusions by selecting one or more intrusions from each class. This technique is known in the software-testing field as *equivalence partitioning* [55]. Ideally, the classes should be selected such that, within each class, either the IDS detects each intrusion, or the IDS does not detect any intrusions [81]. Then, one test case from each class can be selected to represent the class in the final set of test cases. However, in general, it is difficult to identify perfect equivalence classes [81].

Now we consider some possible strategies for classifying intrusions. Intrusions can be classified according to the intrusion technique. A comprehensive example of this type of classification is presented in reference [56]. A second strategy is to classify intrusions based on a taxonomy of the system vulnerabilities that the intrusions exploit (e.g., see [5, 44]). A limitation of using either of these strategies for the purpose of selecting test cases is that, even though two intrusions share the same classification, the IDS might detect one intrusion but not the other. In other words, neither of these classification schemes is likely to produce perfect equivalence classes. However, both of the schemes would ensure that a wide range of test cases would be selected.

A third strategy is to classify intrusions based on their *signatures*, which we described in Section 3.2.4 as encapsulations of the identifying characteristics of specific intrusion techniques. A classification scheme based on signatures is presented in [41]. A limitation of using this classification strategy to select test cases is that the number of classes is small. However, this technique may possibly be extended to yield a finer-grained classification. This technique could also be extended to use information about the internal representation of signatures in a particular IDS.

Given the limitations of the three classification strategies with respect to selecting test cases, the set of test cases should be constructed by using all three strategies. Also, for each strategy, several test cases should be selected from each class. A natural extension to our work would be to develop a large set of test cases for various types of computer systems, which could be used for testing a wide range of IDSs.

As the final step in selecting test cases, the testers can supplement the set of test cases with some simulated intrusions that are of particular interest to the site at which the IDS will be employed. For example, in environments with strict policies governing computer use, some activities that would be considered normal at most sites are considered to be intrusions. The testers can create test cases based on such activities. As a second example, the testers may be aware of intrusion techniques that are not well-known. Simulations of these techniques should be included in the set of test cases.

3.4.3 Limitations of the Methodology

Even thorough testing may not expose all potential weaknesses in an IDS. The primary limitation of our IDS testing methodology is that it tests the IDS against *known* attacks. In many environments, detecting known attacks is valuable, even if the IDS does not detect *any* new attacks. At many sites, known attacks probably occur more frequently than new attacks, because the details of known attacks are often distributed widely (e.g., via scripts that appear in newsgroup postings), while the number of people who know the details of a truly new attack is probably small. However, sites that value security highly are also interested in detecting *new* attacks. If the test-case selection method is thorough, then the IDS should be able to detect new attacks well, because it has been tested against other attacks that are somehow similar to the new attacks. Unfortunately, evaluating the thoroughness of the test-case selection method is an open problem.

IDSs have other potential vulnerabilities. Many IDSs include programs that run on the computers that they monitor. If an intruder can take control of those computers, the intruder can manipulate the IDS programs themselves. Also, if the intruder has knowledge of the database of intrusion signatures in the IDS, then the intruder can attempt attacks that are not represented in the database. Thus, the privacy of the intrusion database should be protected.

3.4.4 Using the Test Results

The test results can be used by the developers, users, and potential customers of an IDS to make the IDS more effective or to make a site more secure. A developer can use the results to find and correct weaknesses in the IDS. For example, if the tests show that the IDS is unable to detect a particular attack, the developer might enhance the language for describing attack signatures, so that the IDS could recognize that attack. Or, if the tests indicate that the IDS is consuming a large amount of resources (e.g., disk space), the developer might create a more efficient implementation that uses less resources. If nothing else, the developer might advertise the weaknesses revealed by the tests, so that users of the IDS can protect their sites by supplementing the IDS with other security tools. An IDS user (e.g., a system administrator) may employ the test results to identify configuration problems, which may occur when the IDS has many configuration options or when the configuration steps are complex. If instead the user detects problems with the IDS itself, then the user can seek additional tools to protect the computer system. Finally, a potential IDS customer can use the test results to compare IDSs and then select the one that will perform best in the customer's computing environment.

3.5 Testing Methodology

We have developed a set of detailed procedures for testing an IDS. The procedures are designed for testing an IDS that monitors a network of computers, although some of the procedures can be directly applied to an IDS that only monitors a single computer. The best environment to use for these tests is an *isolated* local area network, so that computing activity unrelated to the testing can be eliminated.

The installation and configuration of the IDS should be performed carefully. The testers should consult the IDS manuals to determine how to set up configuration files and how to select appropriate values for each configuration parameter[1]. The testing procedures may eventually reveal weaknesses in the IDS configuration, in which case the IDS should be reconfigured and tested again.

Most of our procedures are variations of the following basic testing procedure:

- create and/or select a set of test scripts;
- establish the desired conditions (such as the level of "background" computer activity) in the computing environment;
- start the IDS;

[1]For example, to configure NSM, the user must set up files that indicate the IP addresses of the monitored computers, and the names of the network services to be monitored. In addition, the user must specify in a file a list of strings that NSM should use for pattern-matching against the monitored network traffic.

- run the test scripts; and
- analyze the IDS's output.

We have divided the test procedures into three categories, which correspond directly to the three performance objectives described earlier in Section 3.4.1. Several of the test procedures are adaptations of the "higher-order" software-testing methods described by Myers [55].

3.5.1 Intrusion Identification Tests

The two Intrusion Identification Tests measure the ability of the IDS to distinguish known intrusions from normal behavior. The first of these tests is the Basic Detection Test, which should be conducted as follows:

- create a set of intrusion scripts;
- eliminate unrelated computing activity in the test environment;
- start the IDS; and
- run the intrusion scripts.

The testers can then analyze the IDS output. The specific analysis method depends on the *type* of information available in the output of the particular IDS. We will consider two examples. In the first example, the IDS output classifies each monitored session as "suspicious" or "normal." After conducting the test, the testers can simply calculate the percentage of intrusion scripts that were identified as suspicious.

In the second example, the IDS output consists of a numerical "warning value" for each session, such that a higher warning value indicates a more suspicious session. The testers should compare the IDS output from the test to a large sample of IDS output associated with monitoring normal users in the same computing environment. The testers can use standard statistical techniques to compare the warning values associated with the intrusion scripts to the warning values associated with the normal users. Ideally, the testers should find a statistically-significant difference between the two groups of values. If there is no sample of IDS output available for normal users, then that output can be generated by running normal-user simulation scripts while the IDS is active.

The Basic Detection Test indicates how well the IDS detects intrusions. However, there is a second component in the ability of an IDS to distinguish intrusions from normal behavior. Ideally, an IDS should rarely generate a false alarm by flagging normal behavior as "intrusive." The second Intrusion Identification Test, called the Normal User Test, measures how well an IDS meets this objective. The test is conducted in the same manner as the Basic Detection Test, except that normal-user scripts are used instead of intruder scripts. The IDS output associated with the scripts should be examined to determine how often normal behavior is identified as suspicious. This measurement can be used to estimate how much time will be wasted in investigating false alarms if the IDS is to be used regularly. Returning to the two examples of IDS output described earlier in this section, we note that the Normal User Test is needed only for the first example. The analysis described for the second example, in addition to indicating how well the IDS detects intrusions, should also indicate how often the IDS generates false alarms.

3.5.2 Resource Usage Tests

The Resource Usage Tests measure how much system resources are used by the IDS. The results of these tests can be used, for example, to decide if it is practical to run a particular IDS in a particular computing environment.

At this point, we have developed one Resource Usage Test: the Disk Space Test, which measures the disk space requirements of an IDS. A script that simulates a user who produces computer activity at a constant rate is required for this test. For example, the script might issue a sequence of commands repeatedly.

The procedure for the Disk Space Test is as follows:

- eliminate unrelated activity in the test environment;
- start the IDS;
- run the test script for a measured period of time (e.g., one hour); and
- calculate the total disk space used by the IDS to record the session associated with the script.

The test should be repeated several times using a range of different time intervals. Based on the group of tests, the testers can determine the relationship between disk space usage and monitoring time. For example, in the case of NSM, disk-space usage increases in direct proportion to monitoring time. The tests should also be repeated using different numbers of simulated users, by running copies of the test script simultaneously. Then, the testers can determine the relationship between disk-space usage and the number of users monitored. The testers can use their analysis of all of these test results to predict the IDS storage space requirements when the IDS is monitoring several real users in the real computing environment.

3.5.3 Stress Tests

Stress Tests check if the IDS can be affected by "stressful" conditions in the computing environment. An intrusion that the IDS would ordinarily detect might go undetected under such conditions. We have developed testing procedures for several different forms of stress.

Stress Test: Smokescreen Noise

We define *noise* to be computer activity that is not directly part of an intrusion. An intruder might attempt to disguise an intrusion by employing noise as a smokescreen. For example, an intruder on a UNIX computer might intersperse intrusive commands with normal programming commands, according to this model:

1. with some probability, do an *ls*[2] to check a file;
2. edit a file;
3. compile;
4. with some probability, do an *ls* to check a file;
5. with some probability, go back to Step 1;
6. execute the program; and
7. with some probability, go back to Step 1.

The programming behavior can be used to disguise the *ls* and edit commands which the intruder may be using to examine some target files. Depending on the algorithm

[2] The UNIX *ls* command lists the contents of a file-system directory.

that the IDS is using, the IDS may not detect the intrusive behavior, because the overall behavior appears to be normal.

The first step in the Smokescreen Noise Test is to create suitable test scripts. One approach is to supplement a copy of each intruder script with a sequence of several "normal" commands between each pair of original commands. Then, the test should be conducted like the Basic Detection Test. The IDS output for each script should be compared to the corresponding IDS output from the Basic Detection Test. Ideally, this comparison will show that the IDS detects the same intrusions during each test. On the other hand, if the IDS detects a particular intrusion in the Basic Detection Test, but does not detect the same intrusion in the Smokescreen Noise Test, then that indicates a weakness in the IDS.

Stress Test: Background Noise

We define *background noise* to be noise caused by legitimate user activity. For example, an intrusion may occur during working hours, when there are several legitimate users logged in to the computer system. To prepare for the Background Noise Test, a set of noise scripts that generate continuous normal activity should be created. The scripts for the Normal User Test can be adapted for this purpose.

The testers should start running the noise scripts first. Then the test proceeds just like the Basic Detection Test, in which each attack script is run one at a time. Again, the IDS output for each script should be compared to the corresponding IDS output from the Basic Detection Test, and differences may indicate a vulnerability in the IDS.

The testers should repeat this procedure several times, each time with a different amount of noise. Different numbers of copies of the same noise script (or script set) can be run at the same time to create different levels of noise. If possible, a *maximal* noise level should be used in at least one of the tests. For example, there might be a limit on the number of sessions that the IDS can monitor at the same time. In that case, the IDS should be forced to monitor as many noise-script sessions as possible.

Stress Test: High-Volume Sessions

The Volume Test checks how the IDS is affected by high-volume sessions. The definition of "high volume" depends on the IDS. For example, if the IDS monitors each command in user sessions, then a high-volume session would be a session in which many commands are issued. A "volume script" that simulates a high-volume session should be created for this test.

The purpose of this test is to check if the IDS monitors the high-volume session correctly, and to check if the IDS still correctly monitors other sessions at the same time. This test might detect, for example, a case in which the IDS host runs out of main memory, and so the IDS is physically unable to monitor all of the user sessions at once.

The volume script should be started first. Then, each intrusion script should be run one at a time. After each script has stopped running, the IDS output associated with the volume script should be analyzed carefully. Also, the IDS output associated with each intruder script should be compared to the IDS output from the Basic Detection Test. Differences may indicate that the IDS was affected by the volume script.

This test should be repeated, using different numbers of volume scripts running concurrently each time. As in the Background Noise Tests, a maximal level of volume should be used in at least one of the tests, in which the IDS should be forced to monitor as many volume-script sessions as possible.

Stress Test: Intensity

The Intensity Test checks if the IDS is affected by sessions in which a lot of activity is generated very quickly, and therefore the IDS's information source logs a lot of activity in a short time. First, a "stress script" that simulates such a session should be created. The script should simulate several consecutive user sessions, in each of which the simulated intruder logs in, carries out some intrusive activity, and then logs out. Such a script could be constructed by combining several of the scripts from the Basic Detection Test. *Expect* includes a mechanism that allows the user to specify how quickly consecutive script commands will be issued [47]. Such a mechanism should be used for this stress script so that the commands are issued at a high rate.

The script should be run once. Then, a modified version of the script should be created, which generates the same commands, but at a much slower rate. After the slower script is run, the IDS output associated with the two scripts should be compared. Differences may indicate a weakness in the IDS. For example, due to the high rate of activity caused by the stress script, the IDS might miss some of the intrusive activity.

It should be possible to run several stress scripts concurrently. Then, the stress test can be repeated several times, each time with a different number of stress scripts running. This is important because the IDS may be able to cope with one high-intensity session, but perhaps it will make errors if it is forced to simultaneously monitor several high-intensity sessions. In each case, the IDS output associated with the stress scripts should be compared to the IDS output associated with the slower test script.

Stress Test: Load

The Load Stress Test investigates the effect of the load on the IDS host CPU[3].

This test is conducted in the same way as the Basic Detection Test, except that a high load should be established on the IDS host during the test. A high load can be created by running additional programs on the IDS host, so that the IDS program must share CPU time with the other programs. For a UNIX system, this effect can be enhanced by using the UNIX "nice" command to lower the scheduling priority of the IDS program while other CPU-intensive programs are running on the same host. This tends to decrease the percentage of CPU time allocated to the IDS program by the operating system of the IDS host.

The output from this test should be compared to the output from the Basic Detection Test. Differences may be evidence that the IDS is missing some intrusive activity because it is not running for a high-enough percentage of time on the CPU. This test should be repeated several times, each time with a different load on the IDS host.

3.5.4 Potential Causes of Detection Failures

If the IDS fails to detect a simulated intrusion, the next step in the methodology is to identify the source of the failure. Recall that an IDS that performs misuse detection has two main components: an intrusion database and a pattern-matching mechanism. Also, the IDS depends on another computer system component: the component that logs system activity (e.g., the audit programs). Thus, an IDS may fail to detect an intrusion during

[3] In our testing experiments, we measure the load by using the UNIX *uptime* command, which reports the average number of jobs in the CPU run queue.

testing because:

- the component that logs system activity does not supply enough information to the IDS for the IDS to detect the intrusion;
- the intrusion database does not contain a signature that represents the intrusion; or
- the pattern-matching mechanism fails to recognize a match between a signature in the intrusion database and a record in the system activity logs.

The first problem can be addressed by reconfiguring the logging component, or by adding logging components to the computer system. This might also require changes to the IDS to accommodate changes in the content or format of the information that the IDS analyzes. The second problem is usually easy to correct, because an IDS generally includes a mechanism for adding to its database of signatures. However, the language that is used to describe attack signatures might be inadequate to describe the intrusion. In that case, significant language development work might be needed. The third problem can be addressed by debugging the pattern-matching mechanism.

The testing process should continue until all detection failures are eliminated or at least explained, so that other security mechanisms can be used to cover the weaknesses in the IDS.

3.6 Experimental Results

We conducted testing experiments on NSM. NSM monitors all of the packets transmitted on the LAN to which the NSM host[4] is connected. NSM can associate each such

[4]We use the term *NSM host* to refer to the computer on which the NSM programs are running.

packet with the corresponding computer-to-computer connection. The primary analysis that NSM performs is string-matching; it searches for instances of certain strings in the data streams of the connections. The set of strings is specified in a file by the administrator who runs NSM. The administrator should choose strings that indicate suspicious behavior. NSM assigns a warning value (between 0 and 10) to each connection based on which strings are matched, and on other considerations such as how often a similar connection has occurred in the previous several days. A higher warning value indicates that the activity associated with that connection is more suspicious. We ran NSM on a Sun SPARCstation 2 workstation connected to the Computer Science (CS) LAN segment at UC Davis (UCD). We conducted three rounds of tests: intrusion identification tests, stress tests, and concurrent intrusion tests (described below).

3.6.1 Intrusion Identification Tests

For these tests, we used several different *expect* scripts, each designed to simulate a specific intrusive command sequence. Specifically, the scripts simulate these behaviors:

- browsing through a directory, using the *ls* command to list files, and an editor to view files;
- password-cracking;
- password-guessing using a dictionary file;
- door-knob rattling (password-guessing using common passwords);
- attempting to modify system files (e.g., */etc/passwd*);
- excessive network mobility (moving from computer to computer via *telnet* connec-

tions); and

- exploiting a vulnerability in a system program to obtain *super-user* status.

Each script establishes a *telnet* connection to another computer, sends a sequence of intrusive commands to the remote computer, and then closes the connection. NSM monitored the execution of each of these scripts, and assigned a warning value to each connection. For comparison, we also set up NSM to monitor regular traffic to and from a busy computer on the UCD CS LAN segment for several hours. Although some of this traffic could have been caused by intrusive behavior, we expect that most of it was caused by the legitimate activities of legitimate users.

Ideally, of course, the warning values for connections associated with known intrusive behavior would be high, and warning values for connections associated with normal, benign behavior would be low. Assuming that most of the connections to and from the busy computer were normal, NSM succeeded in this case in assigning a relatively low warning value on average to these connections.

However, NSM also assigned low warning values to some of the connections associated with the intrusive scripts. We determined, though, that this was caused by *our configuration* of NSM. Like many IDSs, NSM can be "tuned" so that it is sensitive to particular intrusive sequences of commands. Our experience illustrates how testing procedures can be used to uncover weaknesses in both the IDS itself *and* the IDS configuration.

3.6.2 Stress Tests

We hypothesized that stress in the form of a high load on the NSM host might affect NSM's ability to monitor network connections. So, we performed a Load Stress Test on NSM, based on the procedure described in Section 3.5.3.

We configured NSM so that it would monitor the TCP (Transmission Control Protocol) packets associated with *telnet* connections to and from a specific computer ("Computer A") on the LAN. To create various levels of load on the NSM host, we used a "load script," which simply creates a *telnet* connection to the NSM host, and then issues a continuous sequence of UNIX shell commands. We measured the load using the UNIX *uptime* command, which reports the average number of jobs in the CPU run queue. We created higher levels of stress in successive tests by running several copies of the load script simultaneously. In addition, we added a second form of stress on NSM. As described in Section 3.5.3, we used the UNIX *nice* command to lower the scheduling priority of the NSM program, so that the operating system would tend to allocate the NSM program a lower percentage of CPU time than it normally would otherwise. We used the *nice* command in the same way for each test.

For each test, we generated the desired load on the NSM host, and then we ran an "intrusion script" on Computer A. The script would establish a *telnet* connection from Computer A to another computer on the LAN, issue a sequence of several intrusive commands to the remote computer, and then close the connection. The intrusion script is a combination of several of the intrusion simulation scripts described in Section 3.6.1.

For each test, NSM produced a report describing the connection established by the intrusion script. Ideally, the report would be identical for each test, because the same script was run for each test. However, the connection reports *were* affected by the increased loads on the NSM host. Apparently, the lower scheduling priority together with high loads on the NSM host caused the NSM program to *miss* some network packets. The NSM connection reports include the number of TCP data bytes missed for each connection. NSM can calculate this number by monitoring the sequence numbers in the TCP headers. As indicated in Figure 3.5, the percentage of data bytes missed by NSM tended to increase as the load on the NSM host increased.

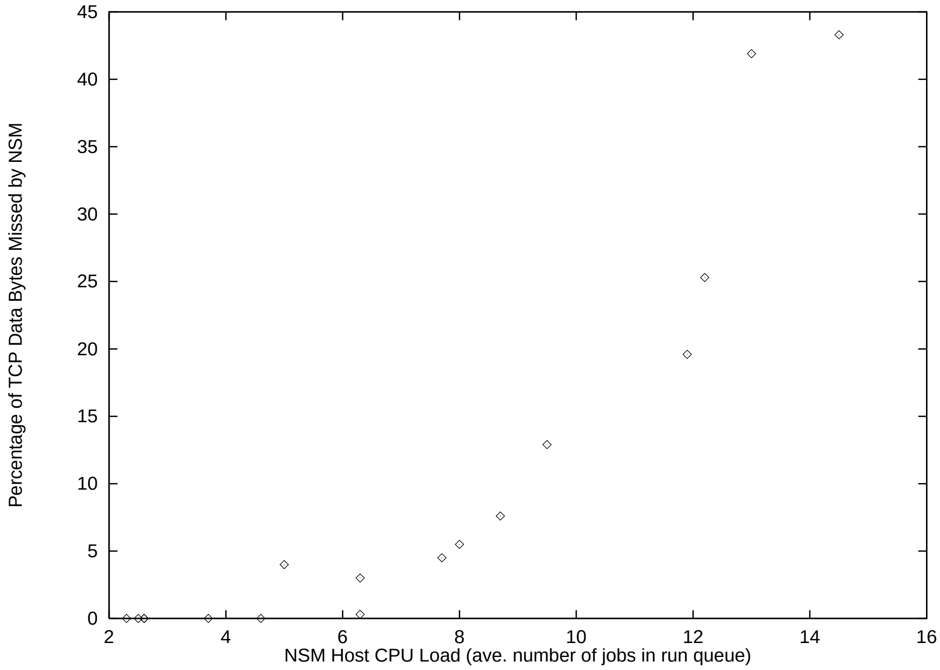

Figure 3.5: Bytes missed by NSM vs. NSM host CPU load.

One possible explanation for the missed bytes is that the NSM program, when it is waiting to run due to the scheduling decisions of the operating system, can miss the transmission of some of the TCP data bytes. However, the monitoring operation involves several components of network hardware and software, and thus the actual explanation for the missed bytes may be more complex. The test results indicate, though, that an IDS can be affected by stressful conditions, and that an intruder might be able to exploit such a weakness to attack the a computer system without being detected.

3.6.3 Concurrent Intrusion Tests

We conducted a third set of tests on NSM, tests that involved simulated concurrent intrusions. We will describe these tests in detail now. To escape detection by an IDS, we hypothesize that intruders might try to distribute their intrusive activity over several concurrent sessions. The premise behind this strategy is that the IDS will assign a higher warning value to one *very* intrusive session than it will to several less-intrusive sessions. We conducted some experiments to test this premise.

We first created a small set of test scripts using our software platform. Each script is designed to simulate a specific intrusive command sequence. Specifically, the scripts simulate the following behaviors:

- transmitting the password file to a remote host;
- password-cracking (via examination of the password file);
- password-guessing using common passwords; and
- exploiting a vulnerability in a system program (*loadmodule*) to obtain *superuser* status.

Next, we created corresponding *concurrent* script sets for each of the same intrusive activities by distributing the intrusive activity over several scripts.

We activated NSM and for each intrusion simulation, we ran the sequential script and then ran the corresponding concurrent script set. We then compared NSM warning values for the sequential scripts with the warning values for the concurrent script sets. The results are displayed in Figure 3.6. NSM assigns a warning value to each network connection. Some of the sequential scripts and all of concurrent script sets initiate more than one network connection, but for clarity the figure shows only the maximum warning value for all network connections associated with each script or script set.

For the simulation of the transmission of the password file, Figure 3.6 shows that the warning value for the concurrent script set is the same as the warning value for the sequential script, and the warning values are high. A possible explanation for this is that the sequential script contains a very suspicious command or set of commands which cannot be divided when the concurrent script set is created. As a result, at least one of the processes related to the concurrent script set is just as suspicious as the execution of the original sequential script. The warning values for the sequential and concurrent versions of the password-cracking simulation are also equal, but in this case the values are lower. A likely explanation is that NSM was not configured to be sensitive to that particular intrusion. So, neither the sequential script nor the concurrent script set produced activity that appeared suspicious to NSM.

For both the password-guessing simulation and the loadmodule attack simulation, the warning value for the concurrent script set is lower than the warning value for the sequential script (dramatically so for the loadmodule attack simulation). In these cases, it was possible to divide up the suspicious commands in the sequential script among two or more processes related to the concurrent script set.

Taken together, our experiments indicate that intruders may be able to reduce their chance of detection by an IDS by distributing their suspicious activities, although this strategy is not always successful. In future work, we plan to investigate the effects of this strategy on different IDSs. For example, while NSM evaluates each network connection independently, some other IDSs evaluate a user's entire history of activity, including the current session and all previous sessions. Against such an IDS, dividing up intrusive activity over several sessions would not be effective, unless each session involved a different user name.

Intrusion Description	Max Warning Value	
	Sequential	Concurrent
Transmitting passwd file	7.472	7.472
Password-cracking	3.160	3.160
Password-guessing	8.722	7.785
Exploiting loadmodule flaw	7.472	4.972

Figure 3.6: NSM concurrent script sets experimental results.

3.7 Conclusion and Future Work

Our experimental results demonstrate that our testing methodology can reveal useful information about an IDS and its capabilities. As the growth in the use and development of IDSs continues, such testing techniques are growing in importance. Future work includes the careful development of a suite of intrusion test cases for the Basic Detection Test. We plan to develop additional performance objectives and tests for IDSs based on the related work of other groups. For example, tests that measure the *processing speed* of USTAT are described in [36]. Another task is to fine-tune the testing procedures and develop suitable metrics to create a "benchmark suite" for IDSs, similar in spirit to the well-established benchmarks such as SPECmarks, Livermore Loops, and Dhrystone, which are used to test the performance of various computer architectures. In the meantime, though, our tools and approaches can be used to systematically evaluate and measure the effectiveness and performance of IDSs. We expect that the development of such methods for assessing IDSs will also have a positive impact on the field of intrusion detection, since developers can use assessment results as feedback in the design process. We also expect that some of our testing techniques can be adapted to testing other software systems. In particular, the stress-testing techniques can be applied to testing computer operating systems and real-time control systems. Also, the software platform can be employed in testing other applications that require simulation of computer users, especially multiple cooperating users.

Chapter 4

WATCHERS

4.1 Introduction

The router is a primary component in the infrastructure of today's Internet, and is therefore an attractive target for attackers. If an attacker can gain control of a router, the attacker can disrupt communication by dropping or misrouting packets passing through the router. Additionally, a router that is simply misconfigured can be disruptive in the same way. We present a protocol that detects and reacts to routers that drop or misroute packets.

The protocol is called WATCHERS: Watching for Anomalies in Transit Conservation: a Heuristic for Ensuring Router Security. WATCHERS protects the routers in an *autonomous system* (AS), which is a set of routers and networks controlled by one administrative authority. WATCHERS is *distributed*; each participating router concurrently runs the WATCHERS algorithm. Each router checks incoming packets to see if they have been routed correctly. Also, each router counts the data bytes that pass through neighboring routers. Periodically, the routers report their counter values to one another, and each router checks if any of its neighbors have violated the *principle of conservation of flow.* This principle asserts that all data bytes sent into a node, and not destined for that node, are expected to exit the node. When a router finds a neighbor that violates the principle, or a neighbor that is misrouting packets, the router stops sending packets to that neighbor. Eventually, the bad router is effectively removed from the network, because all of the bad router's neighbors stop sending packets to it.

In some environments, the conservation-of-flow constraint can be violated by *normal* events. For example, in an AS that supports multicasting, it is routine for a router to receive a single packet and then send out several copies of the packet. WATCHERS cannot be used in such environments yet, but we expect that WATCHERS can be adapted in the future to cope with multicasts and other special events.

WATCHERS has four significant advantages over many other network monitoring techniques:

- A network monitoring tool (e.g., *traceroute* [15] or an implementation of the Simple Network Management Protocol (SNMP) [11]) may fail to detect an attack because the attacker is able to disrupt messages sent by the tool, including messages between separate tool components. WATCHERS uses *flooded transmissions* (see Section 4.1.1) and message authentication to prevent attackers from interfering with communication.
- WATCHERS can detect routers that *selectively* drop or misroute packets, as well as routers that *cooperate* to conceal malicious behavior.
- When they detect suspicious behavior, many network monitoring techniques are un-

able to locate the malicious (or faulty) routers, or they are only able to identify a list of potential suspects [11, 15, 28, 61]. WATCHERS can identify the exact router(s) which are dropping or misrouting packets.

- In ideal conditions we show that WATCHERS never identifies a good router as bad (i.e., never makes a *false-positive* diagnosis). (However, a bad router can sometimes escape detection by WATCHERS: see Sections 4.1.3, 4.4.1, and 4.4.1). We also show how WATCHERS can be tuned to perform nearly as well in realistic conditions.

The rest of this section provides background information on (1) our model of an AS, (2) routing, (3) malicious router behavior, and (4) current router monitoring techniques and their limitations[1]. Section 4.2 presents the WATCHERS protocol. Section 4.3 analyzes the memory, communication, and processing costs of WATCHERS. Section 4.4 demonstrates that WATCHERS is *effective* by discussing WATCHERS' response to several different attack scenarios. Section 4.4 also shows that WATCHERS does not make any false-positive diagnoses when certain conditions hold, and it discusses WATCHERS' limitations. Section 4.5 describes future research tasks.

Significant parts of the work discussed in this chapter were done by students other than the author. First Steven Cheung developed the central ideas [16]. Later Kirk Bradley contributed some new ideas [6]. This author's primary contributions to the WATCHERS project are as follows:

- the complexity analysis discussed in section 4.3;
- the analysis of WATCHERS' effectiveness presented in section 4.4.1;

[1]Readers familiar with these topics may wish to skip to Section 4.2 (our WATCHERS protocol).

- some work on the proof in section 4.4.2; and
- the editing and writing of various WATCHERS documents [7, 8].

After the work discussed in this chapter was completed, another research group at UC Davis did further work on WATCHERS. In reference [34], they discuss some of WATCHERS' weaknesses, and provide some remedies. Their work is not presented in this chapter.

4.1.1 Our Model and Associated Terminology

An *internetwork* is a group of networks connected together so that computers in different networks can communicate. The networks are connected by routers. Each communication between two computers across a network (or internetwork) is divided into segments of data called *packets*. When a packet must travel from one network to another, it is sent to a router. The router receives the packet from one network, and forwards the packet to another network closer to the packet's destination. Each router uses a *routing table*, a matrix that indicates where to forward a packet based on the packet's destination. Each entry in the routing table maps a destination address to the address of the first router on the shortest path to that destination.

The routers and networks in an AS comprise an internetwork. We model an AS as a *directed graph* in which nodes represent routers and edges represent communication links. (Thus a bi-directional physical link should be represented by two edges with opposite directions. For clarity, though, we will represent such a link with a single bi-directional edge.) Routers at either end of a link are called *neighbors*. When a packet travels from one computer to another, it traverses a sequence of zero or more routers (a *route*). A *hop* is a traversal of one link from one router to another.

WATCHERS specifies that routers use *message flooding* to communicate with one another. In this approach, when a router receives a new message, it sends the message to each neighbor except the neighbor from which it received the message. Thus, the message eventually travels over every link and reaches every router in the AS. A router may receive more than one copy of a message, but it ignores the copies after the first [62].

4.1.2 Routing

The routers in an AS communicate with one another using a *routing protocol* to update their routing tables as the network topology changes. Each router in an AS runs the same routing protocol. Many routing protocols fit the following description. Each router monitors the network for topology changes (e.g., temporarily inoperative links) and reports changes to the other routers in the AS via update messages. When a router receives an update message, it recomputes the shortest path to each other router in the AS, and adjusts its routing table accordingly.

4.1.3 Malicious Router Behavior

WATCHERS detects routers that drop packets and/or misroute packets. We refer to such routers as *bad routers*, and we more specifically refer to a router that drops packets as a *network sink*. The simplest network sink drops all packets. It behaves just like an inoperable router, so it can be detected using existing methods (e.g., *traceroute*). However, an *intelligent network sink* drops packets selectively. Some drop packets periodically, while others drop packets based on their contents, including the source and the destination addresses. Still other network sinks consort with other routers to conceal evidence of packet-dropping.

Intelligent network sinks are difficult to detect, since neither the source nor the destination is notified of the location where the packet was lost. Often, the source does not even know the route traversed by the packet before it was lost. Thus, the source cannot pinpoint the location of the network sink.

Instead of dropping packets, a bad router may misroute packets. A router misroutes a packet whenever it forwards a packet to any neighbor other than the neighbor that is closest to the packet's destination (as indicated by the routing table).

Misrouting degrades network service because misrouted packets take longer to reach their destinations. Furthermore, misrouting can cause packets to be dropped. In IP (Internet Protocol) networks, each packet has a time-to-live (TTL) field in its header, which is decremented by each router in the packet's path. Misrouting can cause a packet's TTL value to become zero before the packet reaches its destination, and the IP protocol specifies that packets with a TTL of zero should be dropped. Also, if many packets are misrouted to a specific router, that router may be unable to handle the extra load and may therefore have to drop packets.

WATCHERS cannot detect all malicious router behaviors. In some situations, a bad router can drop or misroute packets, but still escape detection by WATCHERS. For example, a malicious router can drop packets and avoid violating the conservation-of-flow constraint by sending new packets with bogus data to the same destination. (Other examples are described in Sections 4.4.1 and 4.4.1.) In addition, there are bad behaviors that WATCHERS is not designed to detect. A router can send false topology update messages that affect the routing tables of routers throughout the network. Multiple problems may result, including the routing of packets to compromised routers, sub-optimum routes through the network, and isolated routers (i.e., routers that do not receive any packets from their neighbors). A router may also alter or inspect the packets passing through it. Detailed examples of other malicious router behaviors appear in references [3] and [39].

4.1.4 Current Router Monitoring Techniques

Many router monitoring techniques already exist. Here we summarize several of them and comment on their abilities to detect bad routers.

Hop-by-Hop Acknowledgements

Perlman first proposed the idea of intermediate hop acknowledgments [61]. Under this scheme, when a computer S sends a packet to computer D, S receives an acknowledgement (ACK) from D *and* from each router on the path from S to D.

If a packet is dropped on the path from S to D, it appears that the bad router must be either the first router on the path which did not send an ACK to S, or the router previous to that router in the path.

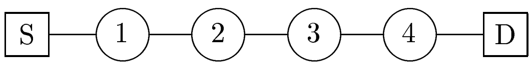

Figure 4.1: A path of four routers between two computers.

However, this security mechanism can be defeated by *consorting bad routers* (routers that cooperate to hide malicious behavior) [61]. In Figure 4.1, suppose router 4 drops a packet (from S and intended for D) while router 1 covers for router 4 by dropping the ACK from router 3. Then it *appears* to S that either router 2 dropped the packet (but sent an ACK) or router 3 dropped the packet (and sent no ACK). Router 1 has shifted the blame away from router 4.

A Probing Technique

Another way to detect misbehaving routers is a probing technique based on peer testing of routers [16]. Using this strategy, a router (the *testing router*) can test its peers along a suspicious route. The testing router sends a *probe* to the first router on the route. A probe is a packet that will follow a pre-determined path, ending back at the testing router. If the probe returns, the next probe is sent through the first (already-tested) router to the second router. If this second probe returns, the next probe is sent through the first and second routers to the third router, and so on until a probe does not return or the destination is reached. If a probe does not return, the router farthest away from the testing router along the probe path is considered faulty or malicious.

While probing should be useful for locating faulty routers, probing does have a weakness: A malicious router can avoid detection if it can distinguish testing probes from ordinary packets.

Network Management

A network management tool monitors the routers and networks in an AS, and can be used to debug problems, control routing, and find computers that violate protocol standards [11, 18]. For a TCP/IP network, such a tool typically runs on a management station and communicates with network elements such as routers using a standard protocol (usually SNMP). The network elements log data and send it to the management tool. The management tool analyzes the data to detect problems. (As described in Section 4.2, WATCHERS operates similarly, except that each router both logs and analyzes data.) A network management tool could be developed to detect routers that drop and/or misroute packets, but we are not aware of an existing tool that matches WATCHERS' capabilities, including its protections against bad routers that attempt to escape detection (see Section 4.4).

Recording and Tracing Routes

Route recording and *route tracing* are often the first methods used to isolate faults in routers [15]. For the Internet Protocol (IP), route recording is optional for each packet. When this option is selected, each router on the path to the packet's destination appends its IP address to a list in the packet's header. However, route recording is a poor way to isolate malicious routers for three reasons. First, the source and destination must agree in advance to use the recording mechanism; if the destination is not informed, it will disregard the route list. Second, the list can be altered by an intermediate router. Third, an intermediate router can simply drop the packet, so that the route list never reaches the destination [18].

Route tracing is similar to probing. The main difference is that probing tests the routers on a pre-selected path between two computers; route tracing identifies and tests the routers in the path that a packet would actually take between the computers. Thus, route tracing reveals the routing decisions of the intermediate routers, and so the method can be used to detect a router that misroutes packets. (The *traceroute* program [15] is a popular implementation of route tracing.)

Route tracing suffers from the same limitation as probing; a bad router might be able to identify a test packet, and thereby avoid detection. Furthermore, since route tracing tests one route at a time, and the number of possible routes in an AS is often large, it is usually impractical to use route tracing to continuously monitor all the routers in AS.

To summarize, existing router monitoring techniques are not adequate for detecting bad routers. Instead, it is necessary to monitor packet flow to detect routers that drop and/or misroute packets. This is the technique that WATCHERS implements.

4.2 The WATCHERS Protocol

This section describes the details of WATCHERS. The goal of WATCHERS is to identify bad routers. Section 4.1.3 defines a bad router as a router that drops or misroutes packets; here we extend the definition to include routers that refuse to participate in the WATCHERS protocol.

4.2.1 Conditions

The following conditions are necessary for WATCHERS to work correctly.

1. *Link-State Condition:* The routers must use a *link-state* routing protocol. In such a protocol, each router is aware of each other router and each link between pairs of routers in the AS. Also, each router periodically broadcasts an update message to

indicate which of its links to other routers are "up" and which are "down" [18].

2. *Good Neighbor Condition:* Every router (good or bad) must be directly connected to at least one good router.

3. *Good Path Condition:* Each good router must be able to send a message to each other good router over a path of good routers. From this condition and the Good Neighbor Condition, it follows that when a router floods a message, it reaches every good router in the AS.

4. *Majority Good Condition:* Good routers must be in the majority. This condition prevents a group of bad routers from prematurely triggering the start of a new round of WATCHERS (see Section 4.2.3).

4.2.2 WATCHERS Counters

Each router counts every packet that arrives or departs over its interfaces. Each router also separately counts arriving packets that have been misrouted.

Data Byte Counters

Henceforth we will refer to the first router in a packet's route as the packet's source, and to the last router in a packet's route as the packet's destination. For any pair of routers, we identify three types of packets for each direction of traffic. Each router counts the number of data bytes in each packet, using a separate counter for each type. (WATCHERS counts data bytes instead of entire packets because packets are sometimes legitimately fragmented into smaller packets by routers. Thus, the number of packets that go into a router per unit time does not necessarily equal the number of packets that come out.) Figure 4.2 illustrates the three types of packets and the associated counters. Consider traffic flowing from X to Y. The $T_{X,Y}$ counter refers to packets that pass through both X and Y. The $S_{X,Y}$ counter refers to packets with source X that pass through Y. The $D_{X,Y}$ counter refers to packets with destination Y that pass through X. The other three counters are similar, but for the opposite direction of flow. Routers X and Y both maintain copies of all six counters to check each other during the WATCHERS diagnosis phase.

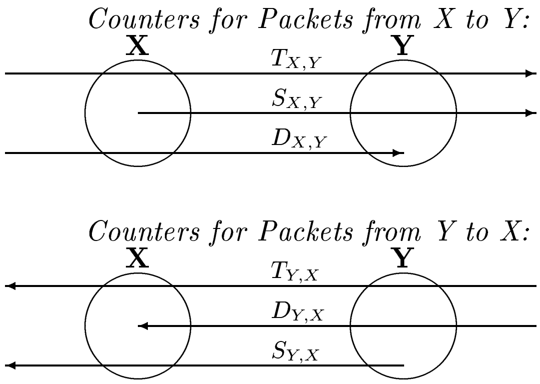

Figure 4.2: Packet byte counters.

Misrouted Packet Counters

Each router also counts the number of times each of its neighbors misroutes traffic. For each neighbor X of router Y, the counter $M_{X,Y}$ records the number of times router X wrongly, with respect to the optimal path, forwards (*misroutes*) a packet to Y. To detect such misrouting, each router maintains a copy of each neighbor's routing table. If an M-counter value is larger than an acceptable threshold (possibly zero), then WATCHERS identifies the associated neighbor as a bad router during the WATCHERS diagnosis phase.

4.2.3 Communication and Diagnosis in WATCHERS

Each router executes the WATCHERS protocol at regular intervals; each execution of WATCHERS is a *round*. In each round, each participating router in the AS runs the protocol concurrently. Each round has two components. First, the *request, receive, and respond sub-protocol* is used to synchronize the routers and to exchange counter values among the routers. Second, each router runs the *validation* and *conservation-of-flow* algorithms to diagnose neighboring routers as good or bad.

Communication: the Request, Receive, and Respond Sub-Protocol

A router participates in the request, receive, and respond (RRR) sub-protocol as follows. First, the router floods a synchronization message (called a *request message*) to indicate a desire to start a new round of WATCHERS. Each request message includes a digitally-signed message digest so that the receiver can authenticate the message's source and contents. The router then waits until it has received a request message from a majority of the routers in the AS. Next the router records a "snapshot" of its counter values, and floods a message containing the values (called a *response message*) to the other routers. Like the request messages, each response message must include a digitally-signed message digest. Neighbors that do not send response messages will be identified as bad during the WATCHERS diagnosis phase. When a router has received all of the counter values it needs, it begins the diagnosis phase of WATCHERS, explained in Section 4.2.3.

The use of flooding for the request and response messages is significant. A bad router might attempt to make a good router appear to be bad by not forwarding a message from the good router. However, the Good Neighbor Condition guarantees that a flooded message will reach at least one good router. From there, the Good Path Condition (together with the action of the message-flooding approach) guarantees that the message will reach every other good router in the AS, even if all bad routers refuse to forward the message.

Typically, a router starts executing the RRR sub-protocol when its clock indicates that it is time to start a new round of WATCHERS. However, a router will also start the RRR sub-protocol when it receives a request message from a majority of the other routers in the AS. Thus, a router with a malfunctioning clock can still synchronize itself with the other routers.

Each AS that runs WATCHERS must have a public key infrastructure to support the WATCHERS authentication mechanism. The infrastructure should include the capabilities to certify and distribute keys. However, a detailed specification of such an infrastructure is beyond the scope of this work.

Diagnosis

The WATCHERS diagnosis algorithm appears in Figure 4.3. The following table explains the notation used in Figure 4.3 and throughout the rest of this chapter.

Notation	**Explanation**
$X \leftrightarrow Y$	X and Y are neighbors.
$A \vee B$	logical *or* of statements A and B.
$A \wedge B$	logical *and* of statements A and B.
$x.T_{x,y}[z]$	x's counter for data bytes in packets with destination z that pass through x and then pass through y (see Sections 4.2.2 and 4.2.3 for further explanation).
$M_{y,x}$	x's counter for packets misrouted by y

The algorithm has three parts: *preliminary checking*, *validation*, and *conservation-of-flow analysis*. **Terminology and Concepts**

A *testing router* is a router running the diagnosis algorithm to test its neighbors (the *tested routers*). A *shared link* is the link between a testing router and a tested router. To test another router, a testing router needs to use its own counters, the tested router's counters, and the counters from each of the tested router's neighbors.

Preliminary Checking

First, the testing router checks if any neighbor has:

- misrouted too many packets;
- not sent a response message; or
- sent more than one different response message.

Each such neighbor is diagnosed immediately as bad. The testing router then performs one more check, but since it is related to validation, we will first explain that phase.

Validation

During validation, a testing router examines its neighbors' counters. Each neighbor's counters for the shared link should match the testing router's corresponding counters.

A discrepancy between any pair of corresponding counters indicates that the tested router is bad. (Small discrepancies can be tolerated, as explained in Section 4.2.5. However, for simplicity of exposition, the algorithm as stated in Figure 4.3 checks for exact matches.)

Algorithm 1 (Diagnosis algorithm, for a given router r)

for each n *such that* $r \leftrightarrow n$
    if $(M_{n,r} \neq 0)$ or
      (n's response message has not been received or not been authenticated) or
      (r has received > 1 *different response messages from n for this round)*
    then r *diagnoses* n *as bad;*
end
for each $(n, t) \in checkSet$
    if $n \leftrightarrow t$ /* *if n and t are still neighbors* */
    then r *diagnoses* n *as bad;*
end
$checkSet = \emptyset$;
/* *Main diagnosis loop: validation and conservation-of-flow analysis* */
for each $d \in$ *destinations in the AS*
    for each n *such that* $r \leftrightarrow n$
        if *(n has not already been diagnosed as bad during this round)* and
          $(r.T_{r,n}[d] = n.T_{r,n}[d] \wedge r.S_{r,n}[d] = n.S_{r,n}[d] \wedge r.D_{r,n} = n.D_{r,n})$ and
          $(r.T_{n,r}[d] = n.T_{n,r}[d] \wedge r.S_{n,r}[d] = n.S_{n,r}[d] \wedge r.D_{n,r} = n.D_{n,r})$
        then
            if $(\forall t \in \{x | x \leftrightarrow n\}$
              (t's response message has been received and authenticated) and
              $(n.T_{n,t}[d] = t.T_{n,t}[d] \wedge n.S_{n,t}[d] = t.S_{n,t}[d] \wedge n.D_{n,t} = t.D_{n,t})$ and
              $(n.T_{t,n}[d] = t.T_{t,n}[d] \wedge n.S_{t,n}[d] = t.S_{t,n}[d] \wedge n.D_{t,n} = t.D_{t,n}))$
            then
              /* *Conservation-of-flow test for neighbor n* */
              $I_n = \sum_{\forall t | t \leftrightarrow n} (n.T_{t,n}[d] \ + \ n.S_{t,n}[d])$
              if $(n \leftrightarrow d)$
                $O_n = (\sum_{\forall t | t \leftrightarrow n} n.T_{n,t}[d]) \ + \ n.D_{n,d}$
              else
                $O_n = \sum_{\forall t | t \leftrightarrow n} n.T_{n,t}[d]$
              if $(I_n \neq O_n)$
              then r *diagnoses* n *as bad;*
            else for each $t \in \{x | x \leftrightarrow n\}$
              if *(t's response message has not been received*
               or not been authenticated) or
                $(n.T_{n,t}[d] \neq t.T_{n,t}[d] \vee n.S_{n,t}[d] \neq t.S_{n,t}[d] \vee n.D_{n,t} \neq t.D_{n,t})$ or
                $(n.T_{t,n}[d] \neq t.T_{t,n}[d] \vee n.S_{t,n}[d] \neq t.S_{t,n}[d] \vee n.D_{t,n} \neq t.D_{t,n})$
              then $checkSet = checkSet \cup \{(n, t)\}$;
        else r *diagnoses* n *as bad;*
    end
end

Figure 4.3: The WATCHERS diagnosis algorithm. Notation: Section 4.2.3.

Each testing router must also check if each neighbor's counters match the counters of *their* neighbors. If a neighbor n has a discrepancy with one of its neighbors t, then the testing router does not perform the next stage of diagnosis (conservation-of-flow analysis) on n. Instead, the testing router assumes that n will diagnose t as bad, or vice versa. The testing router checks this assumption by adding the pair (n, t) to a "check set." In the preliminary checking phase in the *next* WATCHERS round, the testing router checks each pair in its check set to make sure that the pair are no longer neighbors. If n and t are still neighbors, the testing router concludes that its neighbor n is bad, because n and t appear to be cooperating bad routers.

Validation cannot locate all bad routers. Validation identifies routers that drop packets and then change their counters to conceal their behavior. To find bad routers that drop packets but leave their counters intact, we use conservation-of-flow analysis.

Conservation-of-Flow Analysis

In the second stage of diagnosis, a testing router performs conservation-of-flow analysis on each neighbor. For example, if router X in Figure 4.4 is the *testing router*, it performs a test on routers A and B, its neighbors. To test the flow through A, X needs counters from A and each of A's neighbors: C, E, and X itself. To test the flow through B, X needs counters from B, C, F, and X itself.

Conservation-of-flow analysis is based on a simple principle: in any time interval, the number of data bytes going into a router (less the number of bytes destined for the router) should match the number of data bytes that come out of the router (less the number of bytes sourced by the router). The former quantity is the *incoming traffic flow* and the latter quantity is the *outgoing traffic flow.*

For example, suppose X, in Figure 4.4, wants to test A. X can calculate A's incoming traffic flow by using the counters for A's incoming edges (i.e., the counters tracking flow from routers X, C, and E). The expression for A's incoming traffic flow (I_A) is:

$$I_A = \sum_{\forall N | A \leftrightarrow N} (S_{N,A} + T_{N,A})$$

The summation does not include $D_{N,A}$, since this counter is for data bytes destined for A.

Similarly, the expression for A's outgoing traffic flow (O_A) is:

$$O_A = \sum_{\forall N | A \leftrightarrow N} (D_{A,N} + T_{A,N})$$

The counters needed to compute the incoming traffic flow and the outgoing traffic flow are illustrated in Figure 4.5. Now, to test A for conservation of flow, X simply compares A's incoming traffic flow to its outgoing traffic flow. The difference in the two values should be less than the *fault tolerance threshold.*

if $(I_A - O_A) >$ *threshold* **then** X diagnoses A as a network sink;
if $(O_A - I_A) >$ *threshold* **then** X diagnoses A as bad
(because A appears to be artificially generating packets).

The threshold value can be set to 0 for no fault tolerance. (For simplicity of exposition, the diagnosis algorithm as stated in Figure 3 tests for an exact match between the 2 flows.) We discuss thresholds further in Section 4.2.5.

Destination-Specific Counters

The counters described in Section 4.2.2 have one problem: they do not provide enough information to detect consorting routers (see Section 4.1.4).

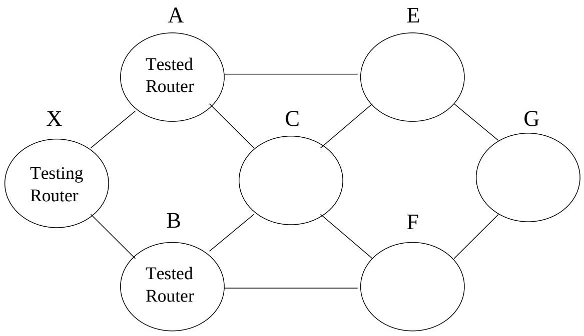

Figure 4.4: A sample router configuration labeled according to testing terminology.

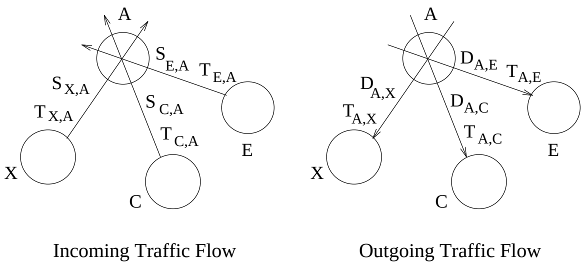

Figure 4.5: Incoming traffic flow vs. outgoing traffic flow: X testing A from Figure 4.4.

For example, suppose that routers 3 and 4 in Figure 4.6 are consorting to drop packets. Assume that the correct route from A to B is $A \longrightarrow 1 \longrightarrow 3 \longrightarrow 4 \longrightarrow B$. However, when A sends a packet to B, the packet travels through routers 1 and 3 (correctly), but then router 4 drops the packet instead of forwarding it to B.

Routers 3 and 4 can hide this attack by incrementing their $D_{3,4}$ counters (instead of their $T_{3,4}$ counters) when the packet reaches router 4. Then conservation-of-flow analysis will not detect the attack.

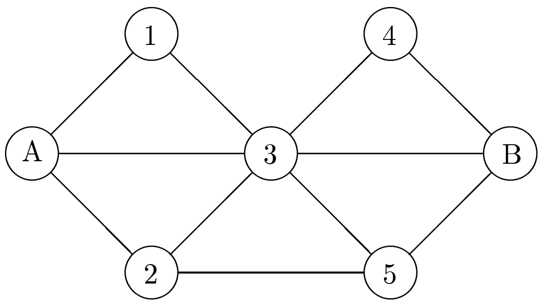

Figure 4.6: A sample router configuration.

To solve this problem, we need to increase the logging requirements. Instead of two S counters and two T counters per neighbor, each router must maintain two S counters and two T counters per neighbor *per destination*. For example, given Figure 4.6, suppose that a packet travels from router A to router B along this route: $A \longrightarrow 2 \longrightarrow 3 \longrightarrow 5 \longrightarrow B$. When router 2 sends the packet to router 3, both routers 2 and 3 should increment their $T_{2,3}[B]$ counters. The $T_{2,3}[B]$ counter is for packets *with destination* B that pass through router 2 and then pass through router 3. (Now we can fully explain by example the notation used for counters in Figure 4.3. The $X.T_{X,Y}[Z]$ counter is owned by router X, and it counts data bytes in packets with destination Z that pass through router X and then pass through router Y.)

Then, instead of one conservation-of-flow check per neighbor, each router must perform one check per neighbor per destination. The flow of data bytes *with a particular destination* into a router must match the flow of data bytes *with that same destination* out of the router. (This explains the first "for loop" in the WATCHERS diagnosis algorithm in Figure 4.3.)

Consider the attack example again, in which router A sends a packet to router B, and the correct packet route is $A \longrightarrow 1 \longrightarrow 3 \longrightarrow 4 \longrightarrow B$. When router 1 sends the packet to router 3, both routers must increment their $T_{1,3}[B]$ counters. When router 3 forwards the packet to router 4, router 4 drops the packet, and both routers 3 and 4 increment their $D_{3,4}$ counters to attempt to hide the attack. However, when router 1 checks the flow of data bytes *with destination* B into and out of router 3, it will find that the number of data bytes coming out is less than the number that went in. Thus, using this new strategy, router 1 *can* detect router 3's bad behavior.

The ability to detect such an attack by consorting routers is an improvement over the conservation-of-flow analysis technique as it first appeared in reference [16].

4.2.4 Response

During a round of WATCHERS, a testing router can diagnose a neighbor as a bad router for six reasons:

1. the testing router did not receive a response message from the neighbor during the

RRR sub-protocol;

2. the testing router received more than one version of a response message from the neighbor;

3. the testing router expected the neighbor to disassociate itself from one of *its neighbors* (based on the previous round of WATCHERS), but the two routers remain as neighbors;

4. the neighbor misrouted a packet;

5. the neighbor failed the validation test; or

6. the neighbor failed the conservation-of-flow test.

For all six cases, the testing router takes the same actions. It floods a routing update advertising the shared link as down (inoperable), which signals the other routers in the AS to remove that link from their network maps. (The bad router may continue to advertise the link as up. However, in at least one link-state routing protocol, the Open Shortest Path First (OSPF) protocol, when two neighbors advertise conflicting information about the status of their shared link, the other routers treat the link as down [52].) Also, the testing router ceases to send packets along that link and acknowledge traffic received on that link. After all of a bad router's neighbors take these steps, the bad router is *logically disconnected* from the network.

4.2.5 Thresholds

Ideally, the data byte counters of neighboring good routers should match perfectly, and for each good router the corresponding misrouting counter values should be zero. In practice, though, the counter values may be different than the ideal values, even for good routers. Therefore, WATCHERS accepts a counter value that is different from the ideal value if the difference is less than a pre-selected threshold.

Three reasons why counter values may be different from the ideal or expected values follow.

- We have implicitly assumed that good routers never drop packets. However, typical network protocols, including several within the TCP/IP suite, drop packets due to errors in transmission or congestion in the network [18]. For these reasons routers with extremely heavy traffic may drop packets at significant rates.

- Neighbors may not be perfectly synchronized when a new round of WATCHERS begins due to propagation delays for the request messages (Section 4.2.3). Assume that there are N routers in the AS, and let $k = \lceil N/2 \rceil$. According to the RRR sub-protocol, each router records a snapshot of its counter values as soon as the kth request messages arrives. We call this instant the *recording time.* The difference in recording times for two neighbors should be less than or equal to the delay on their shared link, since flooding is used to distribute request messages. However, any difference in recording times may cause the counter values of neighbors to disagree.

- Our misrouting detection method depends on neighboring good routers to agree on the network topology. Since a good router sends routing updates whenever it changes its own routing table, and its good neighbors immediately change their own tables accordingly, any period of discrepancy between the routing tables of neighboring good routers should be brief [35]. Moreover, in modern link-state routing protocols, link costs are static and therefore link-state changes are infrequent [75]. Even so, a momentary disagreement over topology may cause a router to increment an M counter with the mistaken belief that a neighbor has misrouted a packet.

The threshold values are also dependent on the WATCHERS *period* (time between consecutive rounds). The period should be adjusted to fit the environment. The period must be at least as long as the time needed by the slowest router to execute one round of WATCHERS. The period should be short enough to avoid counter overflow. Finally, since the threshold values depend on the period, the period and thresholds should be adjusted together.

Setting the thresholds correctly is critical. If the threshold values are too small, too many false alarms may occur. If they are too large, too many attacks might escape detection. However, in some environments, choosing thresholds may be difficult, due to rapidly changing conditions in the network (e.g., network load). Such environments may require threshold values that change dynamically. While we have identified some of the issues involved in setting thresholds, this is an open research problem. Some related work on this problem appears in the intrusion detection literature [73].

4.3 WATCHERS Costs

WATCHERS can only be used if its memory, communication, and processing costs are reasonable. We now demonstrate that these costs depend heavily on the number of routers in the AS and the number of edges in the graph that models the AS (the *router graph*). In addition, the communication and processing costs are proportional to the frequency of WATCHERS rounds. We analyze the worst-case costs for both a "full AS," in which each router has a link to every other router, and for a "sparse AS," which we define to be an AS in which each router has a maximum of three neighbors. We analyze costs in a full AS to show the "absolute" worst case, and we analyze costs in a sparse AS because it is more realistic.

As we have not specified a message authentication system, we do not include the related costs in our analysis. Such a system would add costs in all three categories.

Throughout, we use R for the number of routers in the AS. (R is then also the number of *destinations*, according to our definition of destination in Section 4.2.2.) The number of edges in the router graph is E. The variable N represents the number of neighbors for a given router, and $N = R - 1$ for a full AS. The value of E is $R(R-1)/2$ for a full AS and as large as $3R/2$ for a sparse AS. We use these facts repeatedly in our analysis.

4.3.1 Memory Costs

In the worst case, a router must store $4NR + 3N$ counters: two S counters and two T counters per neighbor per destination ($4NR$ counters), two D counters for each neighbor ($2N$ counters), and one M counter for each neighbor (N more counters). In addition, each router receives a copy of every other router's counters (except for the M counters, which do not need to be shared). Thus each router must be prepared to store $R \times (4NR + 2N) + N$ counters, so the required space is $O(R^3)$ per router in a full AS. For a sparse AS, $R \times (4NR + 2N) + N$ has a maximum value of $12R^2 + 6R + 3$, so the required space is $O(R^2)$.

Each router must also maintain a routing table for each neighbor (see Section 4.2.2). We expect each routing table to contain one entry for each subnetwork in the AS. Since each subnetwork is connected to at least one router, the number of subnetworks is at most cR, where c is the average number of subnetworks connected to each router. We assume that c does not depend on R. Each routing table may also contain entries that indicate where to send packets with destinations outside the AS. To simplify our analysis, we assume that the number of these entries is less than or equal to the number of entries for subnetworks. It follows that the number of entries in each routing table is less than or equal to dR for some constant d. Thus, the space required for a router in a full AS to store a routing table for each neighbor is $O(R^2)$. In a sparse AS, there can be just as many entries in each routing table, but the required space drops to $O(R)$ per router due to the upper bound on the number of neighbors.

4.3.2 Communication Costs

WATCHERS' main communication costs are due to the flooded messages that contain counter values (see Section 4.2.3). As described in Section 4.3.1, the worst-case number of counters in each message is $4NR + 2N$. Each flooded message traverses each link represented in the router graph. Therefore the number of data bits transmitted over all links due to one flooded counter message is proportional to $(4NR+2N)\times E$. Since *each* router floods such a message, the total number of transmitted data bits is proportional to $R\times(4NR+2N)\times E$. Thus the total number of data bits transmitted to exchange counter values is $O(R^5)$ per round of WATCHERS in a full AS. The situation is much better for a sparse AS: $R\times(4NR+2N)\times E$ is at most $R\times(12R+6)\times 3R/2$, which is $O(R^3)$.

Each router also floods a synchronization message during each round of WATCHERS. However, since each such message has a fixed size, the order of growth of the number of bits transmitted per round of WATCHERS is the same as calculated above even if we include the synchronization messages.

WATCHERS' misrouting detection component adds no additional communication costs. The same routing messages that are normally used in a link-state routing protocol are sufficient.

4.3.3 Processing Costs

Next, we divide the analysis of WATCHERS' processing costs into several parts. For the WATCHERS diagnosis algorithm (see Figure 4.3), we consider only the two most time-intensive tasks: validation and conservation-of-flow analysis.

- Routing Table Updates — Each router must compute routing tables for each of its neighbors whenever the network topology changes. For a full AS, this task is equivalent to the problem of finding the shortest paths between all pairs of vertices in a graph. The Floyd-Warshall algorithm solves this problem in $\Theta(R^3)$ time [22]. We compare this cost to the cost for a router that is running a link-state protocol, but is not running WATCHERS. Such a router must update only its own routing table when the topology changes. This task is equivalent to the problem of finding the shortest paths from a single vertex to every other vertex, and can be solved using Dijkstra's algorithm in $O(R^2)$ time [22].

For a sparse AS, a router running WATCHERS can run Dijkstra's algorithm once for each neighbor to update the routing tables, and the running time is still $O(R^2)$, the same as it is for the non-WATCHERS case.

- Counter Upkeep — Incrementing the appropriate data byte counters adds just a small constant to a router's processing time for each packet.

- Message Processing — Each incoming or outgoing WATCHERS message requires some processing by a router, including copying the message data to or from memory, and checking for duplicate messages. We make the approximation that the message processing time is directly proportional to the number of messages that are received or sent, and the size of each message.

  During each round of WATCHERS, each router receives a counter message from every other router, so $R-1$ such messages are received. Each message must be forwarded to each neighbor (except the neighbor from which it was received), so $(N-1) \times (R-1)$ messages are sent. Thus, each router sends or receives $N \times (R-1)$ counter messages during each round. The worst-case size of each message is proportional to $(4NR+2N)$ (as described in Section 4.3.1).

  It follows that the worst-case message processing time is proportional to $N \times (R-1) \times (4NR+2N)$, which is $O(R^4)$ for a full AS. For a sparse AS, $N \times (R-1) \times (4NR+2N)$ is at most $3(R-1)(12R+6)$, which is $O(R^2)$. Each router also floods a synchronization message during every WATCHERS round. Since each such message has a fixed size, including these messages in our analysis would not change the order of growth of the message processing time.

- Validation — For each neighbor, each router checks two S and two T counters per destination, plus two D counters. This amounts to $N \times (4R + 2)$ comparisons, which takes $O(R^2)$ time in a full AS. Each router must also compare each neighbor's counters to the counters of *its* neighbors. If a router has N_1 neighbors, and each of its neighbors has up to N_2 neighbors, then this step requires up to $N_1 \times N_2 \times (4R+2)$ comparisons. Since both N_1 and N_2 can be as large as $R - 1$, these comparisons take $O(R^3)$ time in a full AS. In a sparse AS, the time to check each neighbor's counters is $O(R)$, and the time to check each neighbor's counters against the counters of *its* neighbors is also $O(R)$.

- Conservation of Flow Analysis — When a router X performs a flow test on a neighbor n for a destination d, X must do two addition operations for each of n's neighbors to compute I_n, one addition operation for each of n's neighbors to compute O_n, one operation to check if n and d are neighbors (and if so, one more operation to add one more counter value to O_n), and one operation to compare I_n and O_n (see Figure 4.3). In every WATCHERS round, *for each destination*, each router X must perform such a flow test *on each neighbor*. Assume that X has N_1 neighbors, and that each of X's neighbors has up to N_2 neighbors. The total number of operations necessary is thus as large as $R \times N_1 \times (3N_2 + 3)$. Since both N_1 and N_2 can be as large as $R - 1$, the required time is $O(R^3)$ per WATCHERS round in a full AS. In a sparse

AS, $R \times N_1 \times (3N_2 + 3)$ is at most $36R$, so the required time is $O(R)$.

If the extra processing for the flow tests affects router performance too much, then the tests can be performed by a separate processor. These tests can be delayed without disrupting WATCHERS. On the other hand, routing table updates must be done quickly so that packets are not mistakenly logged as misrouted. Alternatively, the counting of misrouted packets can be suspended until the new tables are ready. Also, a separate processor can compute the new tables so that routing performance is not affected.

Our analysis illustrates that WATCHERS' memory, communication, and processing costs can be expensive for a full AS but reasonable for a sparse AS. In both cases, whether or not WATCHERS is practical depends on the frequency of WATCHERS rounds. The issues related to setting this frequency are discussed in Section 4.2.5.

4.4 Discussion

We claim that WATCHERS is *effective* because it detects bad routers even when they attempt to use various schemes to avoid detection. To support this claim, we describe (in Section 4.4.1) WATCHERS' response to several different types of bad router behavior. We also describe a few scenarios in which bad routers *can* escape detection. In Section 4.4.2, we provide an informal proof of a second claim: in ideal conditions, WATCHERS never diagnoses a good router as bad (i.e., never makes a false-positive diagnosis). In Section 4.4.3, we summarize the limitations of WATCHERS.

4.4.1 WATCHERS' Effectiveness

In each of the following scenarios involving bad router behavior, we assume that the conditions described in Section 4.2.1 hold.

Changing Counters to Implicate Good Routers or Protect Bad Routers

We first consider a bad router that attempts to implicate a good router as bad in the view of another good router.

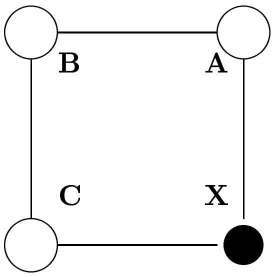

Figure 4.7: A bad router X and good routers A, B, and C.

Consider Figure 4.7. (For clarity, each figure in this section represents only a small part of a typical AS.) Suppose that X changes one of its counters related to the X–A link (e.g., the $T_{X,A}[B]$ counter), so that when B tests A for conservation of flow, B finds a mismatch between the incoming and outgoing traffic flow for A (see Section 4.2.3), and therefore decides that A is bad.

X's scheme fails, though, because before testing A for conservation of flow, B compares A's counters to the counters of A's neighbors. If there are any disagreements, then B does *not* do a conservation-of-flow test on A. Furthermore, when A discovers that its own counters disagree with X's counters, A identifies X as a bad router, and A terminates its neighbor relationship with X. Then, since X is no longer A's neighbor, X cannot even attempt to implicate A as a bad router in subsequent rounds of WATCHERS.

A bad router might try to exploit the fact that a router running WATCHERS does not always do a conservation-of-flow test on its neighbors. Consider Figure 4.8. Suppose that Y attempts to prevent B from detecting that X is dropping packets. Specifically, Y always falsifies its counter values related to the X–Y link, so that they do not agree with X's counters. When B discovers that X's counters disagree with Y's counters, B decides not to test X for conservation of flow, and it appears that Y's strategy has worked.

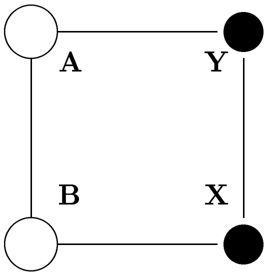

Figure 4.8: Bad routers X and Y with good routers A and B.

However, since X's counters do not match Y's counters, B expects X to terminate its neighbor relationship with Y, or vice versa. When B notices during the next round of WATCHERS that the two routers are still neighbors, B will correctly conclude that both X and Y are bad, and B will terminate its relationship as X's neighbor. The result is the same as it would have been if B detected that X was dropping packets by testing X for conservation of flow.

A bad router might use a variation of Y's strategy in the last example to try to implicate a good router as bad. Consider Figure 4.7. Suppose that X wants B to diagnose

A as bad, so X sends out two versions of its counter values: a correct version to A, and a manipulated version to B (by way of C). To create B's version, X changes its counter values for the X–A link so that they disagree with A's values.

X then expects the following results. B detects the disagreement between the counters of X and A, and expects them to terminate their neighbor relationship. However, since A receives the correct version of X's counter values, A has no reason to end its neighbor relationship with X. When B notices during the next round of WATCHERS that X and A are still neighbors, B suspects that both X and A are bad, and B ends its neighbor relationship with A.

The flaw in X's strategy is that both A and B will receive *both* versions of X's message because WATCHERS messages are *flooded* (see Section 4.2.3). After receiving the correct version of X's message, A will forward it to B, and after receiving the incorrect version, B will forward it to A. Both A and B can recognize that the two versions are different, so both will conclude that X is bad. Thus, X will fail in its attempt to implicate A, and A will terminate its neighbor relationship with X.

In this example, A and B are neighbors. More generally, for any pair of good routers M and N, a bad router will not succeed in sending one version of its counters to M, and a different version to N. By the Good Path Condition (Section 4.2.1), there is a a path of good routers between M and N. Thus, since WATCHERS messages are flooded, any message that reaches M will also reach N, and vice versa. Thus, both M and N will be able to detect that the bad router sent two different messages.

Changing Counters to Avoid Detection

It might appear that a bad router can escape detection after dropping a packet by changing its counter values. We argue now that this strategy does not work against WATCHERS. We start with two observations:

> **Observation 1**
>
> A bad router cannot avoid detection by changing a counter for a link to a good router, because the good router will detect that change during the validation stage of diagnosis.
>
> **Observation 2**
>
> Suppose that a bad router B changes a counter for a link to its bad neighbor C. By the Good Neighbor Condition (Section 4.2.1), B has at least one good neighbor A. If C does not change its corresponding counter, then A will notice that B's counter disagrees with C's counter, and A will expect B to terminate its neighbor relationship with C, or vice versa. If B and C remain neighbors, then A will correctly conclude that both B and C are bad routers, and A will terminate its neighbor relationship with B. Thus, if bad router B changes a counter, then its neighbor *must* change the corresponding counter to prevent B from being diagnosed as bad by a neighbor.

.

Now consider Figure 4.9. Suppose that the bad router X drops a packet, and wants to avoid detection. By the Good Neighbor Condition (Section 4.2.1), X has at least one good neighbor A. If X does not change any counters, then router A will detect the dropped packet when A tests X for conservation of flow.

Therefore X must change a counter. By Observation 1, X must change a counter for a link to a bad router (Y); by Observation 2, X's bad neighbor (Y) must change the corresponding counter. In this way X can escape detection by A. However, now Y has inherited X's original predicament. By the Good Neighbor Condition (Section 4.2.1), Y has at least one good neighbor B. Since Y has changed one counter value, Y will fail B's

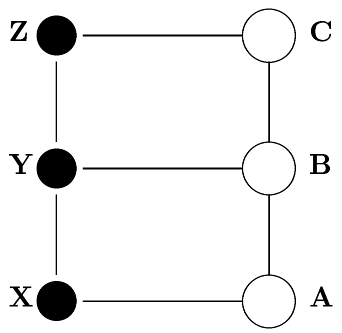

Figure 4.9: Bad routers X,Y, and Z with good routers A,B, and C.

conservation-of-flow test unless Y changes another counter value.

By Observations 1 and 2, Y can only escape detection by changing a counter for a link to a bad router (Z), and that neighbor (Z) must change the corresponding counter. Then Y will avoid detection by B, but Z will inherit Y's predicament.

The pattern is that each bad router involved in the "cover-up" escapes detection only by putting its neighbor at risk for detection. Eventually, time runs out when the counter values for that round of WATCHERS must be reported. At least one bad router will be detected: the router that ran out of time to arrange for its neighbor to cooperate in the cover-up. That router will consequently lose at least one good neighbor. Thus, when a router drops a packet, WATCHERS detects at least one bad router involved in the incident, even though it may not be the router that dropped the packet.

The previous argument holds even when there is a *cycle* of bad routers. Returning to Figure 9, assume that X drops a packet and increments a counter for the X–Y link to hide the drop. In response, Y increments a counter for the Y–Z link. Next assume there is a link from Z to X. Z might protect itself from detection by C by incrementing a counter for the Z–X link. Then, though, X will fail A's conservation of flow test (because A will examine Z's counters). X is back to its original predicament. X can avoid detection by initiating yet another chain of counter changes, but eventually counter values must be reported, and, just as above, at least one bad router will be detected.

Bad Routers at Start or End of Path

WATCHERS does not monitor packet flow from routers to hosts or vice versa. Therefore, WATCHERS cannot detect a packet that is dropped by the first router or the last router in a packet's path.

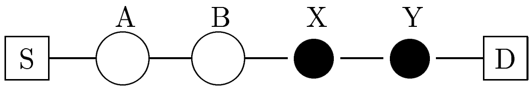

Figure 4.10: Two hosts (S and D); two good routers (A and B); two bad routers (X and Y).

Also, there is an exception to the argument in Section 4.4.1. A bad router that is the next-to-last router in a packet's path can drop the packet and avoid detection, but only if the last router in the path is also bad. Figure 4.10 illustrates this scenario. Suppose that host S sends a packet to host D, and the packet goes through routers A and B to router X. X drops the packet, but still increments its $D_{X,Y}$ counter. Y also increments its $D_{X,Y}$ counter. Then, no router can detect that X dropped the packet instead of delivering it to Y.

We point out this exception for the sake of completeness. However, it is not a significant additional weakness in WATCHERS, because the last router in a packet's path can drop packets without being detected anyway. No collaboration with the second-to-last router is necessary.

Misrouting by Consorting Routers

The misrouting detection mechanism described in Section 4.2.2 can be defeated by two adjacent bad routers. Consider Figure 4.11. Suppose that router A sends a packet to router C, and the correct route for the packet is $A \longrightarrow X \longrightarrow C$. However, when X receives the packet, it forwards the packet to Y. Ordinarily, Y would notice that the packet was misrouted and therefore disclaim X as a neighbor, but here Y is also a bad router and is cooperating with X. Suppose Y forwards the packet to B. To B, Y is behaving correctly, and B is unable to detect that X misrouted the packet earlier. B proceeds to forward the packet to C, and none of the good routers detects X's misbehavior.

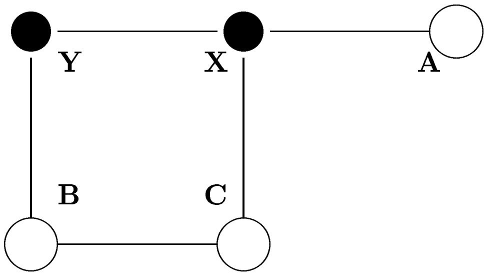

Figure 4.11: Bad routers X and Y with good routers A, B, and C.

However, A *can* detect the misrouting by doing extra analysis on X's packet byte counters. When X forwards the packet to Y, it must increment its $T_{X,Y}[C]$ counter. When A notices that X's $T_{X,Y}[C]$ counter is non-zero, A should then check its copy of X's routing table. In this case, the table will show that X is supposed to route packets that are destined for C to C itself, not to Y. It follows that X's $T_{X,Y}[C]$ counter should be zero. Since the non-zero counter value contradicts the routing table, A can conclude that X must have misrouted at least one packet.

We can generalize this improved approach for detecting misrouting as follows. Each router must test each of its neighbors, and the test can be conveniently incorporated into the WATCHERS validation phase. Assume that a router A is testing its neighbor B. For each of B's non-zero outgoing packet byte counters of the form $S_{B,N}[d]$ or $T_{B,N}[d]$, A must verify that B's routing table instructs B to forward a packet with destination d to neighbor N. If A finds a discrepancy, it can conclude that B has misrouted at least one packet.

4.4.2 WATCHERS: No False Positives in Ideal Conditions

We claim that WATCHERS never makes a false-positive diagnosis if the conditions in Section 4.2.1 hold, and the following conditions also hold:

- *Perfect Transmission Condition:* When any router sends a WATCHERS message to a neighbor, the message arrives intact with no delay.
- *Neighbor Agreement Condition:* Neighboring good routers *always* agree on the network topology.

The following is an informal proof of the claim. There are 6 different events that cause a good router G to diagnose a neighboring router B as bad (Section 4.2.4). We will show that each of these events can occur only if B is a bad router. Therefore, any router diagnosed as bad is indeed bad.

1. *Missing "response" message*: Suppose that G does not receive a "response" message from B during the RRR sub-protocol. Since B and G are neighbors, it follows from the Perfect Transmission Condition (top of Section 4.4.2) that B must not have sent the message. Thus B is bad since it did not participate correctly in the protocol.

2. *More than one response message*: Suppose that G receives at least two different response messages from B during the RRR sub-protocol. (Assume that each message includes a WATCHERS round number, and that in this case, each message is for the current round.) Since WATCHERS includes an authentication mechanism, no other router could have sent the messages. Thus, B is bad because it violated the protocol by sending more than one response message.

3. *Disagreeing neighbors are still neighbors:* Suppose G discovers that B's counters disagree with one of B's neighbors N, but B and N are still neighbors when the next round of WATCHERS begins. Five explanations for the disagreement are possible:

   (a) B's counter values are incorrect, which leads immediately to the conclusion that B is bad.

   (b) N's counter values are incorrect, in which case B is bad for not disclaiming N as a neighbor.

   (c) B's counters match N's counters, but either B's response message or N's response message was corrupted on its way to G. However, the use of the authentication mechanism precludes this possibility; it is not possible for a bad router to change the counter values in a message in transit without destroying the entire message.

(d) B sent two different messages: an incorrect version to G, and a correct version to N. In this case, B is bad for sending two different messages.

(e) N sent two different messages: an incorrect version to G, and a correct version to B. However, by the action of the flooding mechanism, if G received the good message, then its neighbor B must have received that same message. Thus, since B received both versions of N's message, and did not disclaim N as a neighbor, B must be bad.

4. *Misrouted packet*: Suppose G detects that B has misrouted a packet. Then B either disagrees with G on the network topology, or B agrees with G but intentionally misrouted the packet. In the latter case, B is bad because it misrouted the packet, and in the former case, since G is good, B must be bad by the Neighbor Agreement Condition (top of Section 4.4.2).

5. *Validation test fails*: Suppose at least one of G's counters associated with the link between G and B disagrees with B's corresponding counter. The disagreement implies that either G or B is lying. Since G is good, B must by lying, and is therefore a bad router.

6. *Conservation-of-flow test fails*: Suppose G finds that B has violated the conservation-of-flow principle. Two explanations are possible. First, at least one of the counter values that G used in conservation-of-flow analysis may be incorrect. Let S be that set of values. By the action of the detection algorithm, G does not perform flow analysis

on B unless the values in S match B's corresponding counter values. Thus, if at least one value in S is incorrect, then at least one of B's counters must be incorrect, and so B is a bad router. The second possible explanation is that B is dropping packets (or injecting packets) and is therefore a bad router.

4.4.3 Limitations

WATCHERS has these primary limitations:

- WATCHERS does not detect all bad routers. In a few special cases, WATCHERS cannot detect a router that drops packets (as described in Section 4.4.1.) Also, two neighboring bad routers can consort to misroute packets and escape detection (as described in Section 4.4.1). However, Section 4.4.1 also describes how WATCHERS can be enhanced to catch this behavior.
- The Good Neighbor Condition and the Good Path Condition (see Section 4.2.1) are strong requirements, and WATCHERS will not work correctly if they are not fulfilled. However, in an AS of at least moderate size, we expect that most often these conditions will hold if there are only a few bad routers.
- WATCHERS does not detect routers that alter packets. A malicious router can also drop packets yet avoid detection by creating new packets with bogus data and sending them to the same destination (which is equivalent to altering all of the data in the packets). To detect these problems, technologies such as cryptographic authentication mechanisms are needed to complement WATCHERS.

4.5 Future Work

We can identify several potential future tasks, one of which is to implement and test WATCHERS. After implementing WATCHERS, we could measure its memory, communication, and processing costs and compare them to our projections in Section 4.3. We could also investigate the important problems of setting good threshold values for WATCHERS counters, and setting the frequency of WATCHERS rounds. Then we can better evaluate the practicality of running WATCHERS in various environments.

We anticipate that WATCHERS could be extended to include inter-AS packets in its traffic flow analysis. The set of all routers outside the AS could be treated as a single extra node (the *external node*) in the AS graph. Each router in the AS that has a link to any router outside the AS would be considered a neighbor of the external node. Since this approach adds only one node to the AS graph, we project that this extension will not require extensive changes to WATCHERS.

We expect that WATCHERS can be adapted to monitor the collective behavior of a group of several routers (a *supernode*). Each router incident to the supernode would treat the supernode as a single router while running WATCHERS. We predict that it may be less expensive (in terms of memory, communication, and computation) to run WATCHERS in this configuration than it would be to run WATCHERS on each router in the supernode. If the supernode is ever diagnosed as bad, then WATCHERS can be run as usual *inside* the supernode to pinpoint the bad routers. This adaptation may allow WATCHERS to work more effectively in very large AS's.

Finally, we need to enhance WATCHERS to cope with multicasting and other special events which violate the conservation-of-flow constraint during normal operation.

Chapter 5

A Message-Flooding Defense

5.1 Introduction

Server programs on networks are often vulnerable to message-flooding attacks, in which an attacker overwhelms the server with a flood of service-request messages, making the service unavailable to legitimate clients. These attacks can have severe consequences if the service is critical. For example, if a company's World Wide Web server is attacked, the company will lose a significant amount of money if customers are unable to connect to the server and make purchases for an extended period of time. In Simson Garfinkel's article "50 Ways to Crash the Net" [27], at least five of the methods described are message-flooding attacks.

We aim to develop a defense against such attacks. The essence of the defense is that the server regularly inspects its queue of incoming messages and discards messages from unauthorized clients. The defense depends on a strong message-authentication method to detect unauthorized clients.

Section 5.2 discusses the message-flooding problem in more detail and introduces the model client-server environment for which we designed the defense. Section 5.3 describes our proposed defense mechanism. Section 5.4 discusses our simulation of the proposed defense and presents the simulation results. We found that a server employing our defense performs significantly better during our simulated flooding attacks than an unprotected server. Section 5.5 describes a mathematical model for a client-server environment in which the server uses our defense and our analysis of the model. Section 5.6 presents a proof that, under ideal conditions, our defense is impervious to message flooding attacks. Section 5.7 describes related work. Section 5.8 concludes the chapter.

5.2 The Problem

Our defense counters flooding attacks in client-server environments that match the following model. A server, one or more authorized clients, and possibly some unauthorized clients inhabit a network. Clients request service by sending a message over the network to the server. Unauthorized clients might also send service-request messages. The server inserts incoming messages (including messages from unauthorized clients) into a queue. However, if the queue is full when a message arrives, the server discards the message (even if it is from an authorized client).

The goal of the defense is to continue to provide service to each authorized client during flooding attacks.

To detect messages from unauthorized clients, the server must use a message-authentication system. However, even with that, the server is still vulnerable to a flooding attack. An unauthorized client can send a flood of service requests. The server will recognize that these requests are from an unauthorized user, but the server may have to spend so much time on authentication that it cannot maintain good service to legitimate clients. Thus, the server needs to use a fast authentication mechanism.

Still, an authorized client can launch a flooding attack against the server. The client may have turned malicious or it may have a problem which triggers an unintentional flood of messages. To protect against these scenarios, the server needs to employ a fair service policy (e.g., *round robin*). For example, when the server finishes servicing a request from one client, it could then use the policy to select a message out of the queue from a different authorized client.

A fair service policy is not enough, though. If the queue becomes filled with messages from one (misbehaving) client, then the server will have to discard subsequent messages from other clients.

Even if the server uses a fair service policy, an authorized client can still clog up the server's queue with a flood of request messages, preventing messages from other clients from reaching the queue. To prevent this attack, the server needs to monitor the contents of the queue. When a message arrives from a client that already has more than its fair share of messages in the queue, the server should delete that message. Again, a fast authentication mechanism is required. In the event of a flood of service-request messages, the server needs to quickly determine the source of each incoming message, and delete the message if necessary.

5.3 The Defense Mechanism

Based on these observations, we have designed an initial defense against message flooding. The server stores service-request messages in a queue. The queue must have at least as many slots (places where request messages can be stored) as the number of authorized clients.[1] If the queue is full when a message arrives, the server discards the message. If a message is not from an authorized client, then the server discards the message. Also, at any time t, the server must maintain at least one empty slot in the queue for each authorized client with no messages in the queue at time t. If accepting a new message would violate this rule, then the server discards the message. (In practice, the server might accept all messages until the queue is full, but periodically check the queue and discard messages as necessary to adhere to the rule.) The server responds to the request messages in the queue one at a time. Upon finishing a request, the server selects the next request message from the queue using a round-robin strategy. Whenever the server checks for a message from a client and does not find one, the server immediately checks the queue for a message from the next client in round-robin order, unless the queue is empty. The server is idle if and only if the queue is empty.

As our initial observations suggest, this strategy is effective only if the authentication mechanism is fast. (Reference [66] discusses a fast authentication mechanism.) Then, the server can maintain service to authorized clients even during a flood.

[1]For some applications, more than one slot per client should be reserved, if a typical transaction requires more than one request message. However, the Domain Name Service is a prominent example of a protocol in which a typical transaction requires just one request message.

5.4 Simulation

To test our flooding defense strategy, we developed software to simulate a client-server environment like the model described in Section 5.2. We will refer to each iteration of the main loop in the simulation code as a cycle. Once each cycle, the number of new service-request messages from each simulated client is calculated and the simulated server inserts these messages into its incoming message queue if the queue is not full. For each client, the number of messages generated during one cycle of the simulation is a random variable with a Poisson distribution. In our simulation, each client is authorized. We are modeling the scenario in which a malicious authorized client performs the flooding attack. (The impact of a flood caused by an authorized client is similar enough to the effects of a flood caused by an unauthorized client that we found it unnecessary to simulate the latter case.) Thus, the simulated system includes some "good" clients and some "bad" clients." The mean arrival rate of messages from good clients is set low enough that the server would be able to process all messages from all good clients (without discarding any), if the server had only good clients to serve. The mean arrival rate of messages from bad clients is varied to simulate flooding attacks of different intensities. The server periodically checks each message in its incoming message queue. Whenever it finds a message from a client that has more than a pre-selected "cutoff" number of messages in the queue, it deletes that message. When the server is not checking the queue, it is busy processing messages from the queue.

Values for all of these simulator parameters are adjustable:

- duration (number of cycles) of simulation;
- service time (a constant) for each message;
- checking time (a constant): the time required to inspect a single message on the queue;

- delete time (a constant): the time required to delete a single message on the queue;
- number of slots in the queue;
- number of clients;
- number of bad clients;
- mean arrival rate of messages to the server from good clients;
- mean arrival rate of messages to the server from bad clients (henceforth the "flood rate");
- cutoff value; and
- queue checking period (a constant).

All of the time-related values are specified in units of simulator cycles. We do not have performance data from actual servers from which we can estimate the values of the service time, checking time, and delete time. However, our approach depends on the checking time and delete time being much less than the service time, so that messages from bad clients can be dismissed quickly.

Parameter	**Test 1**	**Test 2**
duration	10,000	10,000
service time	1	1
checking time	0.01	0.01
delete time	0.1	0.1
number of slots in the queue	20	20
number of good clients	4	4
number of bad clients	1	1
mean msg. arrival rate, good client (msgs/cycle)	0.2	0.2
flood rate (msgs/cycle)	various: 0-20	various: 0-20
queue checking period	5	various: 1-12
cutoff	various: 4-12	6

Table 5.1: Parameter values for simulator tests.

For our first round of tests, we set the simulator parameters as shown in Table 5.1. We varied the values of two parameters during this round: the cutoff and the flood rate. We first set the cutoff to 4, and then ran the simulator several times, each time with a different flood rate. We repeated this procedure for several different cutoff values between 4 and 12. For each cutoff value, we varied the flood rate from 0 to 20 messages per cycle. The flood rate has a maximum meaningful value due to the simulator design. Specifically, the number of new messages from each client is calculated at the start of every cycle, and no further messages arrive in the middle of a cycle. Since the server simply discards incoming messages when the queue is full, a bad client can fill up the queue but then cannot affect the system further by sending additional messages in the same cycle. Therefore, for each test, the highest flood rate worth investigating is simply Q messages per cycle, where Q is the number of queue slots.

The results are shown in Figure 5.1. We define the Good Service Ratio (GSR) to be the quotient of the number of messages sent by good clients that are eventually processed (instead of discarded or deleted) by the server and the number of messages sent by good clients (regardless of their outcome). The graph displays the GSR versus the flood rate for each of the various cutoffs. As a baseline for comparison, the graph shows the same curve for the case in which our defense strategy is not used.

We interpret the results as follows. When the cutoff is too low (e.g., 4), the server spends too much time deleting messages (particularly when the checking period is relatively short), and too many messages from good clients are deleted. When the cutoff is too high (e.g., 12), the server does not delete enough of the flooding messages, and so there is not enough room in the queue for incoming messages from good clients. For this round of tests, a cutoff of 6 produced the best GSR at the highest flooding rate. No matter what the cutoff,

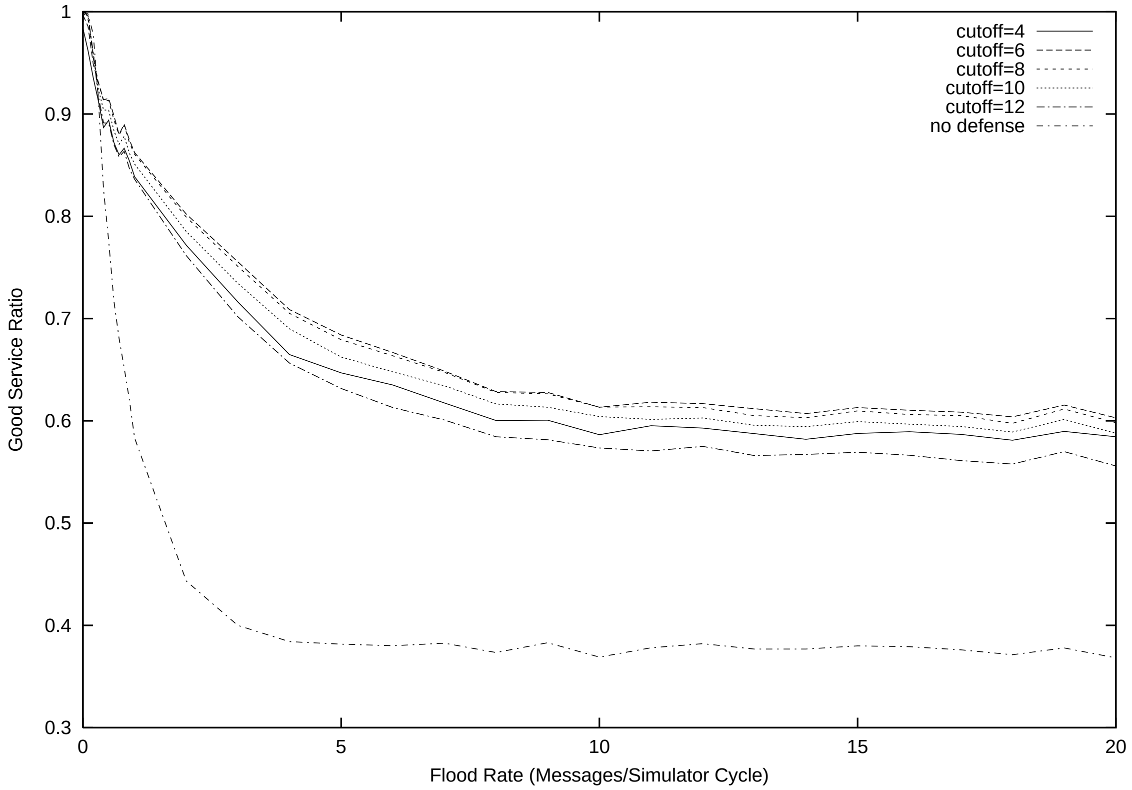

Figure 5.1: Simulation results: various cutoff values.

though, our defense always produced better results than the no-defense case.

The second round of tests was similar to the first, except that we fixed the cutoff value and varied the checking period. For each different checking period, we experimented with several different flood rates. The parameter values for this round of tests are also shown in Table 5.1. The results are shown in Figure 5.2, which displays the GSR versus the flood rate for each of the various checking periods.

The impact of varying the checking period is similar to the impact of varying the cutoff. When the checking period is too short, the server spends too much time checking and deleting (particularly when the cutoff is relatively low). This results in longer average

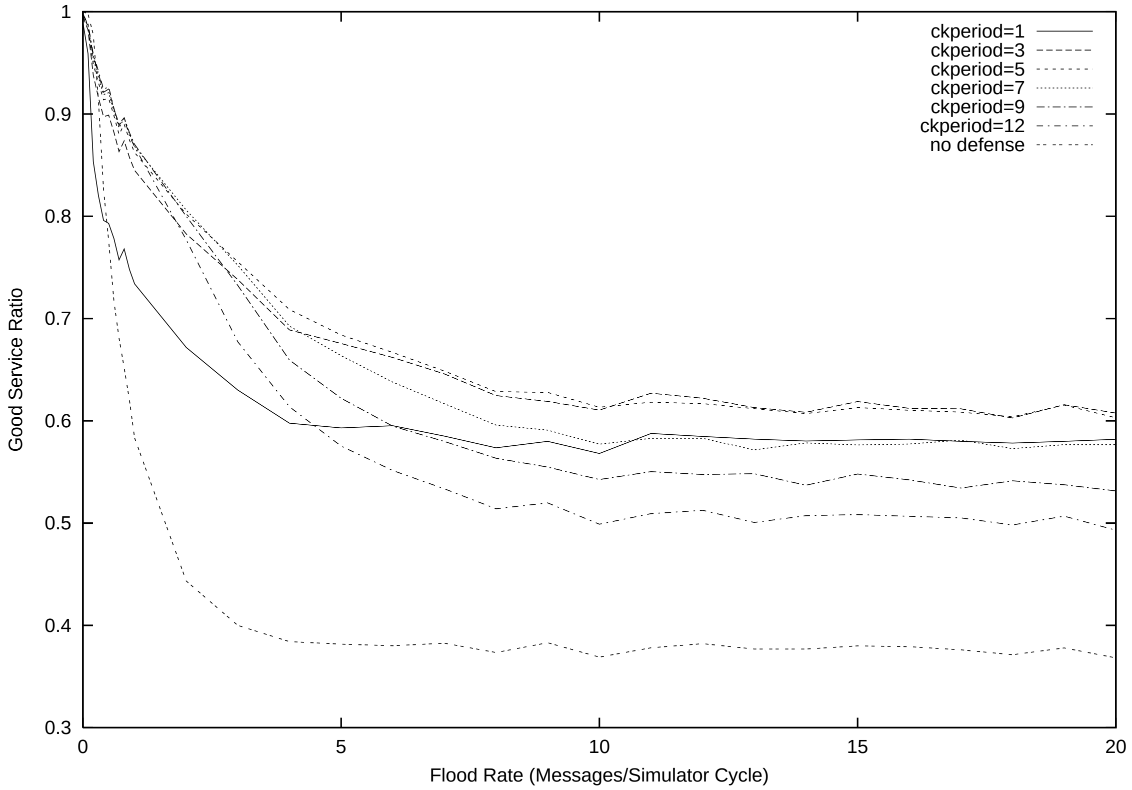

Figure 5.2: Simulation results: various checking periods.

service times, which fills up the queue and causes more messages from good clients to be discarded. When the checking period is too long, the queue tends to stay full for longer periods of time, again causing messages from good clients to be discarded. For this round of tests, a checking period of 3 produced the best GSR at the highest flooding rate. (A period of 5 produced nearly the same results.)

5.5 Mathematical Model and Analysis

In addition to the simulation, we also developed and studied a mathematical model for a client-server environment in which the server uses our defense strategy. The model represents a small system with the following characteristics (defined in Section 5.4):

- service time: 2 seconds;
- checking time: 0.10 second;
- delete time: 0 seconds (a simplifying assumption);
- number of queue slots: 3;
- number of good clients: 2;
- number of bad clients: 1;
- mean arrival rate of request messages from good clients: 1/6 messages/sec;
- mean arrival rate of request messages from bad client (flood rate): 2 messages/sec;
- cutoff value: 1; and
- queue checking period: 0.10 second.

The model server has a small queue (three slots) because the number of states in the model increases rapidly as the queue size increases. The number of clients is limited to the number of queue slots, so that the server can attempt to reserve one slot in the queue per client. As in Section 5.4, the bad client is an authorized client that is misbehaving. (Unauthorized clients are omitted here to simplify our analysis, but they should be added to the model in future work.)

We chose to use a discrete-time Markov chain for the model, shown in Figure 5.3. The discrete-time characteristic makes the model a good match for our simulation, in which events occur only at discrete instants. A Markov chain is appropriate because the environment we are modeling is well-represented by a state machine, and because the Markov property holds for the environment: the probability of any transition to a new state depends only on the current state and not on the history of previous states.

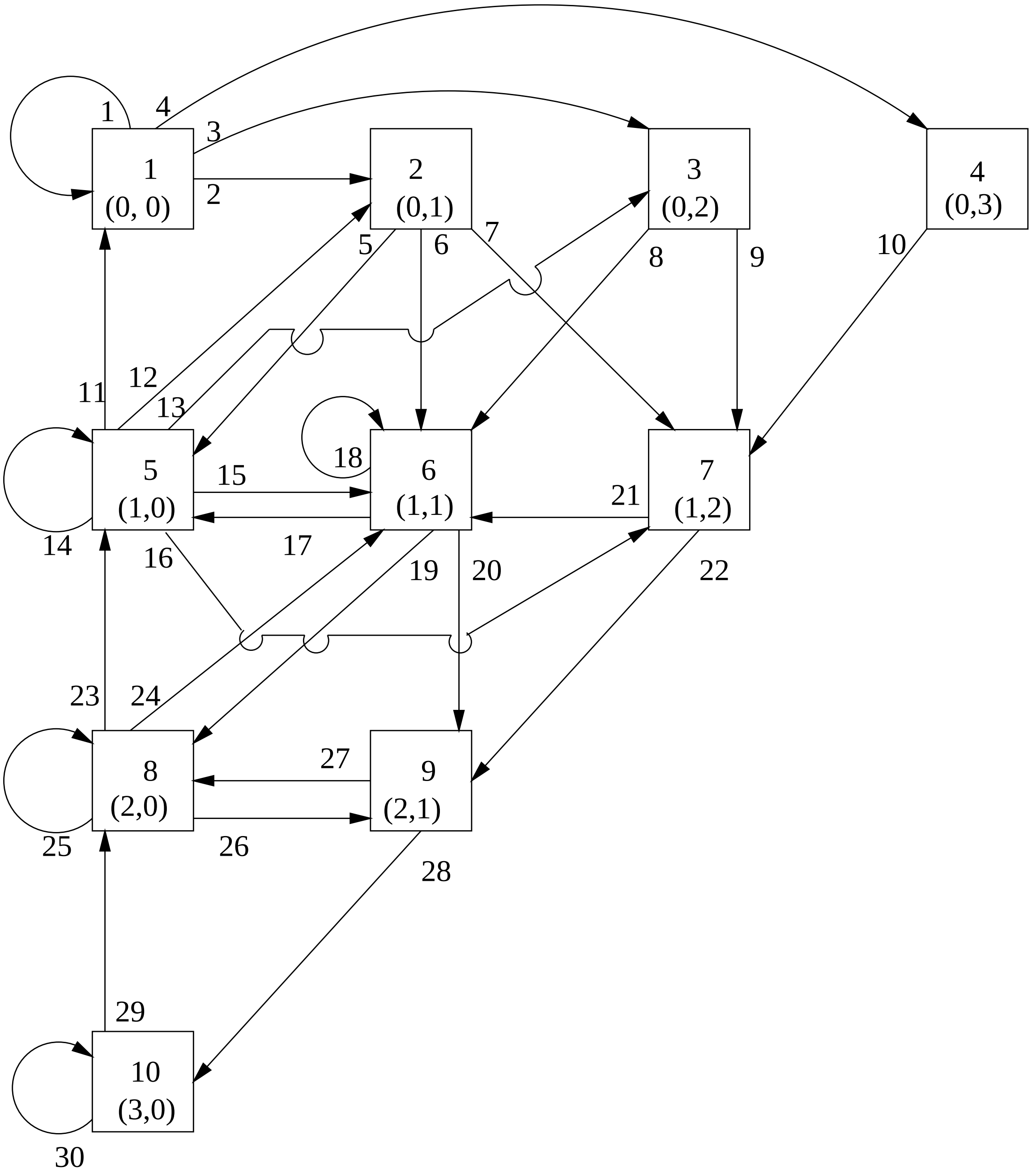

Figure 5.3: The system model.

In Figure 5.3, each box represents a unique state of the server queue. Each state is described by an ordered pair (x, y), where x is the number of messages that have already been checked and accepted by the server and y is the number of unchecked, new messages. When the server checks a new message, it deletes the message if there is another message from the same client already in the queue (because the cutoff value is 1).

Each transition represents one or a combination of the following events:

- a message arriving and being rejected due to a full queue (a "rejection");
- a message arriving without being rejected;
- the server deleting a message from the queue (a "delete"); or
- the server completing work on a request message (a "completion").

In a discrete-time model, events occur only at certain (periodic, in our case) instants of time. We will refer to the interval between any two such instants as a cycle. The duration of a cycle in our model is 0.10 seconds.

The model behaves like the simulated system described in Section 5.4, with some exceptions. First, the cycle duration equals the checking time, while the cycle duration in the simulator is on the order of the (much longer) message service time. Second, the model server checks one new message (if any are in the queue or have just arrived) each cycle, while the server in the simulator may go several cycles without checking. Third, the delete time is zero. These exceptions simplify the model and thereby make it more amenable to analysis.

5.5.1 Analysis

In analyzing this model, our first goal was to determine the state probabilities (i.e., for each state, the probability that the model is in that state) while a message flood is occurring. After that, we intended to develop equations expressing the good service ratio (defined in Section 5.4) and average total time in the queuing system per message (the "system time") in terms of the values of the parameters described in Section 5.4.

We assume that the server is never idle, since the average message arrival rate is 2.33 messages per second, and the server at best can achieve only 0.5 completions per second (because the service time is 2 seconds). We also assume that the system is in steady-state.

Before developing equations involving the state probabilities, we introduce some terminology (taken from reference [54]).

- $S = \{0, 1, 2, ..., 10\}$ is the set of states in the model.
- n is the number of cycles that have passed since the model "started running."
- X_n is the state at time n (i.e., during cycle n).
- For each state i in S, the *state probability* is $\pi_i^n = P(X_n = i)$.
- $\Pi^n = [\pi_1^n\ \pi_2^n\ ...\ \pi_{10}^n]$.
- The *transition probability* of a transition from state i to state j is $p_{ij} = P(X_n = j | X_{n-1} = i)$. (Note that our model is a *homogeneous* chain: the transition probabilities do not depend on n.)
- P is the matrix containing all the transition probabilities (i.e., the element in row i and column j of P is p_{ij}).

In general, $\Pi^n = \Pi^{n-1}P$, for a positive integer n. As a system approaches steady state, though, Π^n converges to Π (and for each state i in S, π_i^n converges to π_i), and then the following equation holds.

$$\Pi = \Pi \times P \tag{5.1}$$

For our model, equation 5.1 actually represents a system of ten equations. We can also immediately introduce one more equation, which follows.

$$\sum_{i=1}^{10} \pi_i = 1 \tag{5.2}$$

If we can determine the transition probabilities (i.e., determine each element of the P matrix), then we can solve Equations 5.1 and 5.2 for the state probabilities. Based on Figure 5.3, each possible transition is listed in the first column of Table 5.2, and the second column shows the events that must occur to cause each transition. Our next step is to determine the probabilities of these events.

It is straightforward to determine the probability of any number of message arrivals at the start of a cycle, because we have modeled the arrival processes for both good and bad clients as Poisson processes for which the mean arrival rates are known. In general, for a Poisson arrival process with a mean arrival rate of λ messages per second, the probability $P_k(T)$ of k arrivals in an interval of T seconds is given by Equation 5.3 below. Table 5.3 shows the calculated arrival probabilities for our model.

$$P_k(T) = \frac{(\lambda T)^k e^{-\lambda T}}{k!} \tag{5.3}$$

We now need to find the probabilities of message departures: rejections, completions, and deletes. We start by defining the following variables:

- X = the fraction of time that the server is servicing request messages. Thus, since the server is never idle by assumption, $1 - X$ is the fraction of time that the server is checking messages in the queue. Also, since the message service time is 2 seconds, $X/2$ = the average number of completions per second.
- Y = the average number of rejections per second.
- Z = the average number of deletes per second.

transition number	required events
1	0 arrivals
2	1 arrival
3	2 arrivals
4	3 or more arrivals
5	0 arrivals
6	1 arrival
7	2 or more arrivals
8	0 arrivals
9	1 or more arrivals
10	a certain transition
11	1 departure and 0 arrivals
12	1 departure and 1 arrival
13	1 departure and 2 or more arrivals
14	0 departures and 0 arrivals
15	0 departures and 1 arrival
16	0 departures and 2 or more arrivals
17	1 rejection and 0 arrivals
18	1 rejection and 1 or more arrivals
19	1 acceptance and 0 arrivals
20	1 acceptance and 1 or more arrivals
21	rejection
22	acceptance
23	1 departure and 0 arrivals
24	1 departure and 1 or more arrivals
25	0 departures and 0 arrivals
26	0 departures and 1 or more arrivals
27	rejection
28	acceptance
29	departure
30	0 departures

Table 5.2: Model transition events.

# of arrivals	probability
0	.7919
1	.1848
2	.0216
3	.0017
4	.0001
more than 4	negligible
1 or more	.2081
2 or more	.0233
3 or more	.0018

Table 5.3: Probabilities: number of arrivals per cycle.

Since messages do not get lost or delayed indefinitely in our model, the average departure rate of messages must be equal to to the average arrival rate, as expressed by the following equation.

$$X/2 + Y + Z = 7/3 \tag{5.4}$$

Next we consider rejections more closely. For example, assume the system is in state 6, with 2 messages in the queue. In one cycle, the expected number of rejections is: $1 \times P(2arrivals) + 2 \times P(3arrivals) + \ldots$. (Fortunately, as Table 5.3 shows, the probability of more than 3 arrivals in one cycle is negligible.)

If PR_i is the expected number of rejections in one cycle when the system is in state i, then Equation 5.5 holds. (Note that the PR_i values can be calculated directly from the values in Table 5.3.)

$$\sum_{i=1}^{10} PR_i \times \pi_i = Y/10 \tag{5.5}$$

The analysis for completions is more straightforward. A completion can occur only if the system is in state 5, 8, or 10, because only in those states is there at least one accepted message *and* zero new messages in the queue. When the system enters one of these states, the server spends the next cycle servicing an accepted message. The probability of a completion at the end of that cycle is the cycle duration (.10 seconds) divided by the service time (2 seconds), which is 0.05 . This leads to Equation 5.6.

$$\sum_{i \epsilon \{5,8,10\}} 0.05 \times \pi_i = X/2 \tag{5.6}$$

The last type of departure we need to consider is the delete. Deletes occur only for states 6, 7, and 9, because only in those states is there at least one accepted message and one new message in the queue. Let AM be the set of accepted messages and NM be the set of new messages in the queue. Also, let src(X) be the set of clients that are sources of at least 1 message in set X. The probability PD_i of a delete while the system is in state i is given by the following equation.

$$PD_i = \sum_{j=1}^{3} P(j \epsilon src(AM) \ and \ j \epsilon src(NM)) \tag{5.7}$$

Here we have reached an obstacle in our analysis which, to date, we have been unable to overcome. Specifically, we have been unable to calculate the probabilities on the right-hand side of Equation 5.7. That problem will require additional work. However, once the PD_i values are known, they can be used in the following equation.

$$\sum_{i \epsilon \{6,7,9\}} PD_i \times \pi_i = Z/10 \tag{5.8}$$

5.5.2 Summary of Analysis

Our first goal was to develop equations for the state probabilities in our model. By assuming that the model reaches steady-state, we were immediately able to produce a system of such equations, expressed by Equation 5.1. The problem with these equations is that their terms include the state transition probabilities, which are also unknowns. Our next step was to investigate these probabilities. Recall that four types of events cause transitions: arrivals, rejections, completions, and deletes. We were able to calculate the probabilities of the first three types of events. It remains for us to find the probabilities for deletes. Knowing the probability of each type of event, we could then determine the probability of each transition in the model (many of which represent a combination of events). Finally, we could then solve Equation 5.1 and Equation 5.2 for the state probabilities.

In our investigation of the transition probabilities, we also defined three new variables (X, Y, and Z), and developed several equations that relate the state probabilities to these variables. Thus, after we solve for the state probabilities, we expect that we can solve equations 5.4, 5.5, 5.6, and 5.8 for X, Y and Z, and use these values to calculate other values of interest such as the good service ratio and average system time. (In fact, for example, the average system time is simply 2/X.)

5.6 Proof: Guaranteed Service in Ideal Conditions

Here, we show that if we make some idealized assumptions about the environment, our defense is impervious to message-flooding attacks. Specifically, no matter how strong the attack, the server will always maintain a guaranteed level of service to authorized, "wellbehaved" (as defined later) clients. The value of this proof is that it forces us to identify the conditions needed for our defense to work and thus reveals the challenges we need to overcome to make the defense work in reality.

5.6.1 The Environment

The environment we consider is the same as described in Section 5.2 and the actions of the server are the same as described in Section 5.3, except that here we need to impose some additional conditions. Next we describe each of the new conditions (and repeat some described earlier for ease of reference).

C1) Messages always arrive instantaneously, and no messages are ever lost or altered in the network, regardless of the level of network traffic.

C2) The service time for each message is exactly T seconds.

C3) At any time t, the server must reserve at least one free slot in the queue for each authorized client with no messages in the queue at time t. If accepting a new message would violate this rule, then the server must discard the message.

C4) Let N be the number of authorized clients. Consider the arrival time t(M) of any message M from any client C. At least until service is completed on M, the server must spend T seconds servicing a message (or messages) from client C in every interval of $N \times T$ seconds starting at time t or after.

For example, the following round-robin strategy meets this condition. After handling a message from client k, the server checks for a message in the queue from client $((k+1)\ mod\ N)$. If there is such a message, the server removes the message from the queue and begins service. Whenever the server checks for a message from a client and does not find one, the server immediately checks the queue for a message from the next client in round-robin order, unless the queue is empty.

C5) For each incoming message, the server *instantaneously* checks for a full queue, checks the new message for the sender's ID, and checks the queue for compliance with condition C3. The server can also discard a message instantaneously. Finally, after servicing a request, the server selects the next message for service instantaneously. Thus, time passes only when the server is servicing a request, and when the queue is empty.

C6) The client must also cooperate to prevent denial of service. We define a "good client" to be an authorized client that never sends two messages M1 and M2 such that $t(M2) - t(M1) \leq N \times T$ (i.e., the interarrival time between 2 request messages is always larger than $N \times T$). (Note that an authorized client is not necessarily a good client.)

5.6.2 Proof

Next, we prove that the server always maintains a guaranteed level of service to good, authorized clients. We start by defining the guarantee and introducing a theorem.

> **Service Guarantee:**
>
> For each request message from a good client, the server will finish servicing the message no later than $N \times T$ seconds after the message arrives.
>
> **Theorem:**
>
> If conditions C1 to C6 hold for the environment described in Section 5.6.1, then the server will always adhere to the service guarantee.

The proof of the theorem is divided into three parts. In each part, we introduce and prove a new claim. By proving each claim, we prove the theorem.

Claim 1:

If a good client C sends a message M at time t1 and that message is accepted and inserted into the queue by the server, and there are no other messages from C in the queue at time t1, then the server will finish servicing M by no later than time t2 = t1 + $N \times T$.

Proof:

Consider the time interval T12 bounded by t1 and t2. Since C is a good client, C will not send another message until after t2. During T12, then, M is the only message in the queue from C. So, by condition C4, the server must spend T seconds on M during T12, and it follows that the server will finish service on M by time t2.

Claim 2:

Let "message n" refer to the nth message sent by client C. We define the Lone-Message Property for each message n from a client C to be true if and only if there are no other messages from C already in the queue when message n arrives. We claim that the Lone-Message Property holds for all messages sent by a good client C.

Proof:

Let S be the set of each natural number n such that the Lone-Message Property holds for message n from a good client C.

When C's first message arrives, there are no other messages from C already in the queue, so the Lone-Message Property holds for message 1, and 1 is an element of S.

Next, assume that k is an element of S. That is, when the kth message from C arrives, there are no other messages from C in the queue.

Let t be the arrival time of the kth message. Since C is a good client, C will not send the next message until after time (t + $N \times T$). By Claim 1, the server will finish servicing the kth message by time (t + $N \times T$). Thus, when message number k+1 from C arrives, the server will have already completed servicing the kth message from C, so the new message will be the only message from C in the queue. Therefore, if we assume that k is an element of S, then (k+1) must also be an element of S.

By the Principle of Mathematical Induction, we conclude that S equals the set of natural numbers. Thus, the Lone-Message Property holds for *all* messages sent by any good client C.

Claim 3:

Whenever a message from a good client C arrives, it is accepted by the server.

Proof:
The server can reject a message only if:

- the message is from an unauthorized client;
- the queue is full when the message arrives; or
- Condition C1 would be violated by accepting the message.

Client C is an authorized client by the definition of a good client. Consider any message M from C arriving at time t. By Claim 2, there are no other messages from C in the queue at time t. By Condition C3, the server must have a free queue slot for client C when M arrives, so the queue is not full at time t. Moreover, if the server accepts M, there will still be the same number of free slots in the queue for clients other than C. Thus, the new state of the queue will not violate Condition C3. We conclude that, when a message from a good client arrives, none of the conditions necessary for the server to reject the message will hold, so the server will accept the message.

5.6.3 Conclusion

Consider any message M sent by a good client C. By Claim 3, M will be accepted by the server. By Claim 2, M will be the only message from C in the queue when M arrives. Therefore, by Claim 1, M will be serviced within $N \times T$ seconds after its arrival.

In summary, then, the server finishes servicing each message from a good client no later than NT seconds after the message arrives, and so the server always adheres to the service guarantee.

5.7 Related Work

Another example of a flooding defense is the "synkill" software tool [68] that protects against a SYN-flooding attack. First we will describe the attack. To establish a TCP connection, three packets are required: a connection request from the client, a reply from the server, and reply from the client. This sequence is commonly called a "three-way handshake." To perform the attack, the attacker sends a connection request packet with a spoofed source IP address to a server. The server sends a reply to the spoofed address, but that address is "inactive:" No computer on the Internet is ready to receive packets to that address. The server then waits for the client reply, which will never arrive. The connection is in a "half-open" state because the handshake is incomplete. The problem is that often the server stores data regarding half-open connections in a queue, and the size of the queue is limited. When the queue is full, the server can no longer accept connection requests from legitimate clients. Eventually, the server will give up on half-open connections and clear space in the queue, but the attacker can then fill up the queue again.

The synkill tool counters the attack by monitoring the server's local network and creating a database of IP addresses observed in connection request packets. If a request arrives from address N but the final packet in the handshake never follows, then synkill classifies N as "bad." As soon as another connection-request packet from N arrives, synkill sends a "reset" packet to the server so that the server will quit on the connection immediately. Thus, it will be more difficult for the attacker to fill up the half-open connection queue.

The problem with synkill is that it can be defeated by an attacker that changes the (bogus) IP address of each successive connection request during an attack in a way that synkill cannot predict. However, the authors [68] point out that one of the published attacked scripts used the same bogus IP address for every packet.

The synkill approach suggests a way to adapt our approach to a different environment, an environment in which clients do not need to be authorized and every client has legitimate access to the server. The server would create a database of source addresses found in service-request messages and monitor the frequency of messages from each address. If the frequency of a particular address were to exceed a threshold, that address would be classified as bad. The server would perform checking and deleting as described previously, except that it would consult the database to make decisions about which packets to delete. This approach would suffer the same weakness as synkill (described above). An advantage of this approach, though, would be that a strong authentication method would not be required, which would reduce checking costs.

5.8 Conclusion and Future Work

Our tests show that our defense mechanism can help a server perform significantly better during a flooding attack. An attractive feature of the defense is that it requires changes to server-side software only.

In Section 5.6, we identified the idealized assumptions required for our defense to work. Future work includes research on the appropriate technologies to determine how close to these ideals we can come. For example, we need to study the speed of strong authentication mechanisms. We also need to implement the defense for an existing server program, so that we can measure typical values of parameters such as checking time and delete time.

Additional future work includes considering optimizations. For example, the defense could be invoked only when a flooding attack is recognized. Also, we expect that the defense could be implemented as a program that functions as a filter for an existing server program. That is, incoming messages would first go to the filter, which would implement the message queue and the checking and deleting functions. The filter would send accepted messages to the real server program. Then, neither the client nor the server software would require modifications (though some modifications might be necessary for communication between the filter and server).

Chapter 6

Grammar-Based Partition Testing of Network Programs

6.1 Introduction

Software problems in network programs can create security vulnerabilities (e.g., see Section 2.2.2). Such vulnerabilities can be prevented by thorough software testing. We have developed a systematic methodology for testing a network program. The methodology is an example of *partition testing*, in which a "program's input domain is divided into subsets, with the tester selecting one or more element from each subdomain" to be test cases [81]. The partitioning in our methodology is based on analysis of the grammar that specifies valid incoming messages to the program. The goal of the methodology is to establish high confidence in the program under test. The methodology might reveal some software problems, too, but after these are corrected, an administrator can run the program knowing that it is unlikely to have a security vulnerability since it has passed rigorous tests.

We used this methodology to test a Unix "finger daemon" (*fingerd*) program. We discovered one flaw that, depending on the configuration of the finger service, could lead to a denial-of-service attack. However, the flaw can be easily corrected, and otherwise the program exhibited no security weaknesses during our tests.

Part of our testing procedure is to create a preprocessor, a program which performs a few simple checks on an incoming message and then either discards it or delivers it to the network program. (For example, the preprocessor could check the length of each message.) The main purpose of the preprocessor is to make testing more practical by limiting the variety of incoming messages that the network program needs to handle. After testing, the preprocessor should always be used to protect the network program.

Section 6.2 provides an overview of our testing methodology. Section 6.3 describes the purpose of the preprocessor in our approach, and the details of a fingerd preprocessor. Section 6.4 explains how we analyzed the grammar in the finger protocol specification (RFC 1288 [83]) to develop test cases. Section 6.5 describes the software system we developed to perform the tests. Section 6.6 describes the test results. Section 6.7 discusses a second round of testing we did on a fingerd program to demonstrate that, without our methodology, thorough testing of a fingerd program seems impractical. Section 6.8 describes related work. Section 6.9 concludes this chapter and describes future work. Appendix A contains a list of our test cases.

6.2 Overview: Testing Methodology

An important goal in testing a network program is to establish convincing evidence that the program can handle all possible incoming messages without malfunctioning in a way that could cause a security problem. Our methodology is one approach toward achieving this goal.

The methodology has the following steps. First, a preprocessor is designed to filter some messages with invalid formats. This filtering simplifies testing, as explained in Section 6.3. Next, the set of all possible messages that might still reach the network program is partitioned into several subsets based on the grammar that specifies valid incoming messages. For example, Figure 6.1 shows the grammar for valid messages to fingerd (according to RFC 1288 [83]). The subsets are selected such that the network program is expected to behave similarly for each message in any one subset. One or more test cases are selected to represent each subset, where each test case is simply a message to the network program in the format corresponding to its subset. Finally, all the test-case messages are sent to the network program and the program's response to each message is observed.

As stated in reference [59], partition testing is a standard testing approach. Our contributions include the application of this approach to testing network programs and the use of the grammar that specifies valid inputs (incoming messages) to guide the partitioning.

Our testing approach is a "black-box" approach: test cases are selected based on the program specification. The alternative is "white-box" testing, in which test cases are selected based on knowledge of the program itself. We believe our black-box approach is appropriate because we expect that often the users of network programs either do not have

```
{Q1}    ::= [{W}|{W}{S}{U}]{C}

{Q2}    ::= [{W}{S}][{U}]{H}{C}

{U}     ::= username

{H}     ::= @hostname | @hostname{H}

{W}     ::= /W

{S}     ::= <SP> | <SP>{S}

{C}     ::= <CRLF>

(SP represents a space character, and CRLF represents
a carriage return followed by a linefeed.)
```

Figure 6.1: The Unix finger daemon query grammar.

access to source code or do not have the resources to analyze the code. Also, even when both of these conditions are false, black-box testing can identify some software problems that white-box testing cannot detect. For example, if a program fails to check for a condition on the input described in the specification, black-box testing will reveal the failure (presuming the specification was studied carefully during test-case selection), while white-box testing may not.

6.3 The Preprocessor

The first step in our testing methodology is to develop a preprocessor. A preprocessor is a program that performs some simple checks on an incoming message, discards the message if it fails any of the checks, and otherwise forwards the message to the network program. We assume that whenever the network program is in service, the preprocessor will be used to filter messages to the program. Thus, the preprocessor simplifies the testing of the network program, because test cases can be selected from the set of messages that the preprocessor will allow to pass, instead of from the (much less constrained) set of all possible messages that could be sent to the network program.

Of course, the preprocessor must be prepared to receive any message from the latter set. It may appear at first that we have simply shifted a difficult testing problem from the network program to the preprocessor. However, we expect preprocessors to be short programs with limited functionality. Therefore, we further expect that testers can convince themselves of the correctness of a preprocessor with only a modest testing effort.

Without a preprocessor, thorough black-box testing of a network program would often be impractical. Consider our experimental configuration, in which a query to a computer running fingerd is delivered in the data portion of a single network packet, and the maximum length of that data string is 1500 bytes. Thus, the length of the username, for example, in a legal query can be anywhere between 1 and 1498 bytes (2 bytes are needed for the carriage return and linefeed at the end). Thorough testing requires that we partition this large range into several intervals. Then, whenever we select a test case query that includes a username, we will actually have to create several test cases, each one using a username from a different interval. The problem becomes more severe when a test case query includes more than one component (e.g., username *and* hostname) with a large range of potential sizes or values. The number of test cases quickly grows large. In summary, the preprocessor imposes additional constraints on valid messages which ultimately reduce the number of test cases required for thorough testing.

We have developed a specification (Figure 6.2) for a fingerd preprocessor.[1] The first acceptance criterion in the specification comes from RFC 1288. The next two are based on our experience with typical user names and host names in the Unix environment. (These values can be changed without affecting the methodology described here.) The fourth constraint contradicts RFC 1288, which permits any number of "@" tokens. However, we believe it is a reasonable constraint, based on our experience with using the finger protocol. For a small loss in functionality, we reduce the number of test cases considerably. The preprocessor also converts some character strings to others to further reduce the number of test cases.

The number of elements in the set of strings that the preprocessor will allow to pass to fingerd is still large. However, the limits established by the preprocessor (e.g., maximum length of username) make it easier to partition this set into subsets.

6.4 Partitioning Strategy

We will refer to the set of all queries accepted by the fingerd preprocessor as the "input set." Here, we describe how we partitioned this set in order to select test cases.

We started by considering legal queries, where a legal query is one that adheres to the grammar described in Figure 6.1, and an illegal query is one that does not. The grammar allows seven categories of queries:

- empty queries (containing just a newline character);

[1] This initial version covers query string analysis correctly, but does not address the communication details between the preprocessor and fingerd, and between the preprocessor and inetd (the Internet daemon, the program that should invoke the preprocessor).

Finger Daemon Preprocessor Specification

- input:
  - pointer to start of query data;
  - length of data; and
  - total length of buffer containing data.
- output:
  - accept flag: 1 or 0 (set according to acceptance criteria described below);
  - new length of data; and
  - initial query data is revised as described below.
- revisions to data:
  - maximum query length: 322 characters. (This number is based on the longest string that still meets the acceptance criteria described below.) Longer queries are truncated.
  - character/string substitutions:
    - /W $->$ /w
    - CRLF $->$ $\backslash n$ (carriage return, linefeed to newline)
    - CR $->$ $\backslash n$ (carriage return to newline)
    - LF $->$ $\backslash n$ (linefeed to newline)
- acceptance criteria:

  1. each byte in the query string must represent an ASCII character;
  2. longest alphanumeric string permitted before first '@' (i.e., longest username): 8 characters;
  3. longest alphanumeric string permitted, after an '@' has been scanned (i.e., longest hostname): 30 characters.
  4. maximum of 10 '@' characters per query; and
  5. after the first '@' character, every alphanumeric string must be preceded directly by an '@' character.

Figure 6.2: Fingerd preprocessor specification.

- queries containing only "/w";
- queries of the form "/w username";
- queries starting with "@hostname";
- queries starting with "username@hostname"; and
- queries starting with "/w @hostname"; and
- queries starting with "/w username@hostname".

We first partitioned the input set into subsets according to these seven categories. Next, we considered each subset one at a time, and attempted to subdivide further. (The first two subsets contain just one query each, and cannot be subdivided.) For example, consider the "/w username" format. One characteristic that can vary among elements of this subset is the length of the username. Thus, we chose to divide this subset into three groups:

- username length = 1 character (minimum length);
- username length = 2 to 7 characters; and
- username length = 8 characters (maximum length).

Our choice was based on a software testing maxim: if a variable can potentially assume any value in a range, test the minimum value and the maximum value, because the program may react differently to these values than it would to other values in the range.

A second characteristic is the validity of the username (i.e., does the username in the query correspond to a user account on the computer?). Note that an invalid username can still be part of a legal query. Using this characteristic, we divided the three subsets described above into six.

A more thorough test might attempt to subdivide further. For example, another potential characteristic is whether or not the string contains any numbers. Yet another is the position in the query string in which the first number appears. These characteristics could be used to subdivide query subsets further if the tester decided that they are likely to make a difference in the processing of queries by fingerd. The drawback of subdividing is that it increases the number of test cases, since at least one test case will be selected from each subset. This is especially true if combinations of characteristics are considered to be significant. For example, for the two characteristics considered above (length and validity of user name), we created subsets for each combination of values for the two characteristics. A different strategy would have been to create one subset for each of the choices of length and just one subset for invalid user names. The implicit assumption behind that strategy is that the program handles all invalid user names the same way, irrespective of the length of the name. (We took the conservative approach by choosing *not* to make that assumption.)

After partitioning the set of legal test cases, we used the same approach to partition the set of illegal test cases. We first created three subsets, because a fingerd query (which has been accepted by the preprocessor) can be illegal for three different reasons:

- it contains a bad character (one that does not appear in any legal token),
- it contains an incomplete token; or
- it has one of these token-related problems:
  - at least one missing token;
  - a token of the wrong type in at least one position; or
  - at least one extra token.

We believe that this strategy for partitioning the set of illegal test cases is particularly significant because it can be immediately applied to the testing of any software for which a grammar is used to specify valid input. Its scope is likely broader, too, because even a software specification without an explicit grammar for valid input often still implicitly defines such a grammar.

We decided that an interesting characteristic of an illegal test case is whether or not it contains a newline character, because the fingerd program might wait to receive the newline character before it processes the query. (Every legal query must end with a newline character, as indicated by Figure 6.1.) Thus, we divided each of the three subsets above into two, based on presence or absence of a newline in the query.

Most of the other characteristics we used to partition the three main subsets of illegal test cases were related to the position of the first occurrence of the problem in the query string.

In summary, we partitioned the input set manually, and then selected 194 legal test cases and 78 illegal cases. Each test case is represented by an input file (created manually) for our *front* program described in Section 6.5. It would have been challenging to automate both the partitioning and the test-case selection, because both processes seem to require some judgement that is difficult to express as an algorithm.

6.5 Testing Software and Procedure

After creating the test cases, we developed a small software system to send the test-case queries from one computer to another computer running the fingerd program.

We first developed a short program called *front* (for "front-end"), which outputs a string of data based on the contents of an input file. The input file can contain any number of lines; each line starts with a code that instructs *front* how to convert the rest of the line to values in the output data buffer. (The exception is code 3, which simply signals the end of the input file.)

Code	Meaning
0	Insert following number into buffer.
1	Insert newline character into buffer.
2	Insert following text string into buffer, followed by a newline.
3	End of input file.
4	Insert following text string into buffer, without a newline.
5	Read the following number n, then insert n instances of the following character into buffer.
6	Read the following number n, then insert n instances of the following text string into buffer.
7	Insert a space into buffer.
8	Insert following text string at end of buffer.
9	Insert following text string plus a newline at end of buffer.
10	Read following number n, then insert n random characters into buffer.
11	Read following numbers n and m, then insert n instances of m into the buffer.
12	Read following numbers n and m, then insert into buffer a string of n values starting with m and increasing by one each time.

Table 6.1: Codes for *front* input file.

The second component in our system is the *netcat* utility [32], which can be configured to read data from standard input, store the data in a TCP packet, and then send the packet to a specified host and TCP port number.

We combine the two components using the Unix pipe mechanism. For example, if "case1" is the front input file that represents test case 1, we could send our test case query to the TCP finger port (79) on the computer named "targethost" with the following command (where % represents the shell prompt).

% front case1 | nc targethost finger

The final component in our software system is a Unix shell script, which issues such commands for each of the test cases.

6.6 Test Results

We used our software system to send the test-case queries listed in Appendix A to a target host running fingerd.[2] The target host was a personal computer running the FreeBSD operating system, version 3.3.

We examined each of the responses. We were testing for security problems only, so we did not check the accuracy of each response. Instead, we looked for warning messages on the console of the target host concerning unusual fingerd behavior. We also looked for cases in which fingerd did not reply to a query, because the lack of a reply might indicate that the query caused fingerd to "crash."

None of our queries caused fingerd to crash. We received a reply to each query (many correctly containing error messages such as "host not found") except for queries missing terminating newline characters. For each of these queries, fingerd appeared to get stuck waiting to receive more data, as suggested in Section 6.4. The waiting period ended only when we broke the connection from the client side. This problem could enable a denial-of-service attack. An attacker could send several such queries at about the same time. In the worst case, the target computer would spawn a fingerd process for each request, and the computer's resources would be consumed by these useless processes. In the case of the FreeBSD operating system on our target host, though, the finger service was configured to limit the number of simultaneous fingerd processes to three. (We checked one other operating system, SunOS 5.7, and found that it has a similar configuration option.) Thus, the most harm that an attacker could do with this method would be to prevent legitimate

[2]We had to modify some of the test cases. For example, we decided not to create new hostnames, so we were unable to use a valid hostname with just one character in the test cases. Instead, we substituted an existing hostname with two characters.

users from using the finger service.

Fortunately, we could easily enhance the preprocessor to check for a missing newline. Then, the fingerd program we tested, running in conjunction with the enhanced preprocessor, would be able to handle each of our test cases without signs of a security vulnerability. This result should increase the confidence of system administrators who need to decide whether or not running this version of fingerd is a security risk.

6.7 Testing a Fingerd Program for Errors

In Section 6.3, we stated that thorough testing of fingerd without a preprocessor seems impractical. The number of test cases required seems to be very large, since several of the test-case query components have a large range of possible values. Thus, we hypothesized that existing versions of fingerd may have software problems that could be revealed by testing. In particular, we expected that "extreme" test cases might reveal these problems, as more ordinary test cases (or their equivalents) would have been used by developers in their own testing of their finger programs.

To test our hypothesis, we developed a second set of test cases. Most of the test cases are variations of our original test cases, with large numbers of characters and/or tokens. For example, to test the "/w user" format, our test cases include user names of various lengths, starting with 32 characters, then doubling each time up to 2048 characters.

Due to the much larger set of potential queries, our test-case selection was more ad hoc than it was for the first round of tests. We did not attempt to "cover" the set of potential queries by partitioning the set and then selecting representative test cases from each partition. Instead, we simply aimed to develop a set of extreme test cases with some variety. This change reflects a change in testing goals. Initially, we aimed to develop a set of test cases such that, if the fingerd program passed each test, then a system administrator could be confident that running fingerd was not a security risk. In the second round of tests, we were attempting to find errors, not trying to establish that fingerd has no weaknesses that could cause security problems.

We performed the second round of testing in the same way as the first: our software system sent each test-case query, one at a time, to the target host running fingerd. We examined each response and also watched the target host's console for fingerd-related warning messages. We used approximately 25 test cases for this round of testing.

One of the test cases did reveal a software problem. The query contained a username token ("puketza") followed by a space, repeated 47 times. The target host did not reply and we found a warning message on its console indicating that the program invoked by fingerd attempted to use a bad address.

We do not know if this problem causes a security vulnerability. We can only speculate that if the data in a query can cause a program to attempt to access a bad address, then perhaps a carefully crafted query can cause the program to overwrite values in the stack, as in a buffer overflow attack [23].

This result supports our hypothesis that flaws in current versions of fingerd exist because it is impractical to test a fingerd program thoroughly unless the set of potential query strings is constrained.

6.8 Related Work

Our testing methodology is similar to (and inspired by) the "category-partition" method (CPM) defined by Ostrand and Balcer [59]. In this method, the testers also identify characteristics of program inputs and "environment conditions." (An environment condition is a condition related to the *state* of the computer system. For example, whether or not a username is valid is an environment condition in CPM terminology.) These characteristics and conditions are called "categories." The testers then identify one or more potential values for each category, called "choices." (For example, we identified length of username as a characteristic and then selected three choices of values for that characteristic.) Each choice describes a subset of the possible inputs and environment conditions.

A test case is generated by selecting a choice from each category. To create the entire set of test cases, all possible combinations of choices are generated, subject to constraints that prevent some choices from being combined with others. For example (adapted from an example in the CPM paper), if a program requires a string input argument, then the choice "omitted" is mutually exclusive with the choice "several embedded spaces".

Our approach is more flexible than the CPM in one sense. As described in Section 6.4, we *first* partitioned the set of incoming messages to fingerd into two subsets: legal and illegal queries. We then divided the set of legal queries into seven subsets and the set of illegal queries into three subsets. Only then did we start to use query characteristics to partition these ten subsets in a manner similar to CPM. The advantage of our initial partitioning is that many of the characteristics do not apply to all ten subsets. For example, the "location of first bad character" characteristic applies only to the subset of queries containing at least one bad character. CPM includes a method for specifying that characteristics apply only to certain subsets, but the method appears to be cumbersome for a large number of such characteristics. Partitioning the set of all possible queries into several subsets of similar queries, and then applying the CPM approach simplifies test-case selection.

6.9 Summary and Future Work

We have developed a methodology for testing network programs. The methodology is based on analyzing the grammar that specifies the format of a valid incoming message to the program. We used the methodology to test one version of the Unix fingerd program and found that it had a weakness that, depending on configuration, could lead to denial-of-service attacks.

An important part of our methodology is the development of a preprocessor program. The preprocessor is a filter for the network program and so it reduces the size of the set of potential messages that might reach the network program. This reduction make thorough partition testing of the network program feasible. The preprocessor can also be used to "patch" weaknesses in the network program, such as the one described above.

We hypothesized that current versions of fingerd, tested by developers not using our methodology, are likely to have software problems because the size of the set of potential incoming messages is large enough to make thorough testing impractical. To check our hypothesis, we developed some "extreme" test cases, and we did discover one flaw in a current version of fingerd that might lead to a security problem.

For future work, we could attempt to apply our testing approach to other network programs. Our approach is applicable if the protocol that the program implements includes a grammar for the format of incoming messages. In particular, we could try to use our approach to test programs that implement more comprehensive protocols with more complex grammars, such as HTTP.

Chapter 7

Conclusion

We have used three basic approaches in our computer security research: filtering, testing, and detection. This chapter summarizes our work related to each approach, discusses specific contributions, and considers future work.

7.1 Summary

We developed a design and prototype for a filtering device called Safe Modem (Chapter 2) that protects a single computer from Internet threats, much like a firewall protects a group of computers. Safe Modem intercepts and examines network packets, and discards those that are suspicious or potentially dangerous.

Regarding testing, we developed a methodology for testing and evaluating intrusion detection systems (IDSs) (Chapter 3). We identified a set of desirable qualities for an IDS and then designed detailed testing procedures to measure an IDS's performance with respect to those qualities.

Concerning detection, we helped to develop a protocol called WATCHERS (Chapter 4) that detects routers that drop or misroute packets. The protocol checks for "conservation of flow": the number of data bytes in packets flowing into a router in a given time interval should match the number of data bytes in packets flowing out of the router. Each router also checks each incoming packet to see if its neighbor routed it correctly.

In further filtering work, we developed a filtering strategy that protects network programs against message-flooding attacks (Chapter 5). The essence of the strategy is that the server scans its queue of incoming messages periodically, searching for messages from unauthorized clients and excessive messages from misbehaving authorized clients. The server quickly discards such messages to make room in the queue for messages from legitimate clients.

As a second example of testing research, we developed a methodology for testing network programs (Chapter 6). The essence of the methodology is that we select test cases based on analysis of the grammar that specifies valid incoming messages to the network program. We used the methodology to test an instance of the Unix finger daemon program (fingerd). The goal of the methodology is to select a set of test cases such that, if the program responds correctly to each test, a system administrator can have high confidence that the program does not have a security weakness.

7.2 Research Contributions

We made the following specific contributions to the security research community.

- **Safe Modem:** We developed a software prototype with two key security functions: e-mail filtering and header-based packet filtering. The same technique that we used to implement e-mail filtering can also be applied to filtering other protocols such as FTP. Furthermore, the modularity in both our design and prototype facilitate adding other analysis components.

- **Testing Intrusion Detection Systems:** We made three key contributions with this work. We described several strategies for the difficult task of selecting test cases for testing IDSs. We described several IDS testing procedures in detail, especially procedures related to stress testing. We also developed a software platform that facilitates the simulation of users for testing.

- **WATCHERS:** We analyzed the complexity of WATCHERS' memory and communication requirements and running time. We made WATCHERS' strengths and limitations more clear by analyzing WATCHERS' response to several potential attack scenarios. We also contributed to a proof that, given certain idealized conditions, WATCHERS never misdiagnoses a good router as bad.

- **Message Flooding Defense:** Our work provides a straightforward strategy for protecting a server against flooding attacks. The strategy is applicable to any setting in which the server has a set of authorized clients, but can receive messages from unauthorized clients, too. The defense can be implemented inside a new server program or it can be implemented as a filter which protects a server program.

- **Network Program Testing:** One useful by-product of our work is a good set of test cases for the Unix fingerd program. More important, we expect that the methodology we used to test fingerd can be applied to several other network applications (though we need to test applications more complex than fingerd to confirm this expectation).

7.3 Future Work

Each of our projects can be extended.

- **Safe Modem:** Analysis components should be developed for other significant protocols, especially FTP and HTTP. Safe Modem could be implemented in several different ways; these approaches should be studied and compared. Also, we expect that some components of Safe Modem (e.g., the "string matcher") can be implemented in hardware and thereby accelerated. Some hardware research will be required to check on this possibility.

- **Testing Intrusion Detection Systems:** Future work includes the careful development of a large set of test cases for testing IDSs. Then, our methodology should be applied to several different IDSs. This process should naturally lead to refinement of the testing procedures and to identification of more desirable characteristics of IDSs. The set of test cases together with our testing procedures could be enhanced into a "benchmark suite" for IDSs, similar in spirit to the well-established benchmarks for testing the performance of various computer architectures.

- **WATCHERS:** WATCHERS has now been implemented (by a different research team at UC Davis [34]). The implementation should be used for measurements of WATCHERS memory, communication, and processing-time requirements. It can also be used to investigate how to set good tolerance threshold values in WATCHERS, to prevent too many false alarms. This is important for enabling WATCHERS to work effectively in real environments. We have also considered adaptations that would allow WATCHERS to monitor more than one autonomous system and monitor very large autonomous systems.

- **Message-Flooding Defense:** Our defense depends on strong and fast message authentication. Thus, we need to investigate the performance of existing authentication mechanisms. We also need to implement our approach, perform some experiments, and then compare the results to our simulation results.

- **Testing Network Programs:** Our testing approach includes the development of a preprocessing filter that reduces the number of test cases required for the network program. We have assumed that the preprocessor, because it is limited in its functionality, can be thoroughly tested with only modest effort. We need to check this assumption by creating and testing a fingerd preprocessor. We also need to apply our approach to an application that is more complex than fingerd, such as an HTTP server.

Appendix A

Test Cases for Partition Testing of Fingerd

The following test cases were used in the testing procedures described in Chapter 6.

```
# Notes:
#  - Each test case appears on a separate line, after the test case
     number.
#  - A "#" character indicates that the rest of the line is a comment.
#  - There are 3 sets of test cases:
#       - legal queries
#       - illegal queries (with character-related problems);
#       - illegal queries (with token-related problems);
#    The 3 sets are numbered separately.
#  - Notation:
#      - The string "..." is used to represent repetition of some pattern.
#      - A character followed by a number X in parentheses indicates
#        repetition of the character X times.
#  - Except where noted, the last character of each query is a newline
#    character, which is not shown in most of the test cases below.
#    In some cases, the newline is indicated explicitly by the string
#    "\n" within the query. In a comment, a newline is indicated by "NL."
#  - Hostnames:
```

```
#       - valid hostnames:   c, c1 through c10, c(30), k6
#       - invalid hostnames: d, d1 through d10, d(30)
#       - valid but unreachable hostnames: e, e1 through e10, e(30)
#   - Usernames:
#       - valid usernames:   a, aa, a(8), puketza
#       - invalid usernames: b, bb, b(8)

# Legal Queries
    # "empty string" format
        1           # actual query is a newline character only
    # "/w" format
        2 /w        # just one instance of this format
    # "/w user" format
        3 /w a          #shortest valid username
        4 /w puketza    #valid username with length > min and < max
        5 /w a(8)       #longest valid username
        6 /w b          #shortest invalid username
        7 /w bb         #invalid username with length > min and < max
        8 /w b(8)       #longest invalid username
    # "@hostname" format
        9 @c            #shortest valid hostname
        10 @c1          #valid hostname with length > min and < max
        11 @c(30)       #longest valid hostname
        12 @d           #shortest invalid hostname
        13 @d1          #invalid hostname with length > min and < max
        14 @d(30)       #longest invalid hostname
        15 @e           #shortest unreachable valid hostname
        16 @e1          #unreachable valid hostname with length > min
                        # and < max
        17 @e(30)       #longest unreachable valid hostname
        # queries containing multiple "@" characters
          # queries containing only valid hostnames
            18 @c1@c2                  #2 different hostnames
            19 @c1@c2@...@c9@c10       #10 different hostnames (max
        #  number of @'s)
            20 @c1@c1                  #same hostname, twice in a row
            21 @c2@c2...@c2            #same hostname, 10 times in a row
          # queries containing only invalid hostnames:
            22 @d1@d2                  #2 different hostnames
            23 @d1@d2@d3@d4...@d10     #10 different hostnames (max
                                       # number of @'s)
            24 @d1@d1                  #same hostname, twice in a row
            25 @d2@d2...@d2            #same hostname, 10 times in a row
          # queries containing only unreachable valid hostnames:
```

```
          26 @e1@e2                    #2 different hostnames
          27 @e1@e2@e3@e4...@e10       #10 different hostnames
          28 @e1@e1                    #same hostname, twice in a row
          29 @e2@e2...@e2              #same hostname, 10 times in a row
        # strings of valid hostnames (of length N) followed by 1
# invalid hostname
          30 @c@d                      #N=1
          31 @c1@c2...@c9@d            #N=9
        # strings of valid hostnames (of length N) followed by
        #  one unreachable valid hostname
          32 @c@e                      #N=1
          33 @c1@c2...@c9@e            #N=9
  # "user@hostname" format
      # Repeat all test cases for the "@hostname" format twice,
      # but add to the start of each query:
      #   - (cases 34-58) a valid username (for the first round); and
      #   - (cases 59-83) an invalid username (for the second round).
      # Repeat all test cases for the "/w user" format three times,
      # but in each query drop the "/w" and add to the end
      # an "@" followed by:
      #   - (cases 84-89)  a valid hostname (1st round);
      #   - (cases 90-95)  an invalid hostname (2nd round); and
      #   - (cases 96-101) an unreachable valid hostname (3rd round).
  # "/w @hostname" format
      # (Cases 102-126) Repeat all test cases for the "@hostname"
      # format, but add "/w " to the start of each query.
  # "/w user@hostname" format
      # (Cases 127-194) Repeat all tests for the "user@hostname" format,
      # but add "/w " to the start of each query.

# Illegal Queries
  # queries containing at least 1 bad character:
    # queries without a newline:
      # first bad character appears:
        # at start of string
            # rest of string forms legal query (except for missing NL)
                1 !/w puketza@k6
            # string ends immediately after bad character
                2 !
        # just after incomplete token at start of string
            3 /!
        # just after one good token at start of string
            # rest of string forms legal query (except for missing NL)
                4 /w! puketza@k6
```

```
        # string ends immediately after bad character
            5 /w!
    # just after 1 good token and a space
        # rest of string forms legal query (except for missing NL)
            6 /w !puketza@k6
        # string ends immediately after bad character
            7 /w !
    # just after an otherwise legal (except missing NL) query
        8 /w puketza@k6!
    * just after an otherwise legal (except missing NL) query
      and a space
        9 /w puketza@k6 !
    * at last possible position after an otherwise legal
      (except missing NL) query
        10 /w puketza@k6 ... !       #255 spaces between '6' and '!'
# queries containing a newline:
  # first bad character appears:
    # (before NL)
      # at start of string
        # rest of string forms legal query
            11 !/w puketza@k6
        # string ends immediately after bad character
            12 !
      # just after incomplete token at start of string
            13 /!
      # just after one good token at start of string
        # rest of string forms legal query
            14 /w! puketza@k6
        # string ends immediately after bad character
            15 /w!
      # just after 1 good token and a space
        # rest of string forms legal query
            16 /w !puketza@k6
        # string ends after bad character
            17 /w !
      # just after an otherwise legal query
        18 /w puketza@k6!
      # just after an otherwise legal query and a space
        19 /w puketza@k6 !
      # at 2nd-to-last possible position after an otherwise legal
      #  query (NL is in the last position)
        20 /w puketza@k6 ... !\n     #255 spaces between '6' and '!'
    # (after NL)
      # just after an otherwise legal query
```

```
          21 /w puketza@k6\n!
        # just after an otherwise legal query and a space
          22 /w puketza@k6\n !
        # at last possible position after an otherwise legal query
          23 /w puketza@k6\n ... !     #255 spaces between NL and '!'
# queries containing at least 1 incomplete token:
# (Note: the only possible incomplete token in this case is "/")
  # queries without a newline:
    # first incomplete token appears:
      # at start of string
          # rest of string forms legal query (except missing NL)
              24 / puketza@k6
          # string ends immediately after incomplete token
              25 /
      # just after one good token at start of string
          # rest of string forms legal query
              26 /w/ puketza@k6
          # string ends immediately after incomplete token
              27 /w/
      # just after 1 good token and a space
          # rest of string forms legal query
              28 /w / puketza@k6
          # string immediately ends after incomplete token
              29 /w /
      # just after an otherwise legal query
          30 /w puketza@k6/
      # just after an otherwise legal query and a space
          31 /w puketza@k6 /
      # at last possible position after an otherwise legal
        (except for missing NL) query
          32 /w puketza@k6 ... /      #255 spaces between '6' and '/'

  # queries with a newline:
    # first incomplete token appears:
      # (before NL):
          # at start of string
              # rest of string forms legal query
                  33 / puketza@k6
              # string ends immediately after incomplete token
                  34 /\n
          # just after one good token at start of string
              # rest of string forms legal query
                  35 /w/ puketza@k6
              # string ends immediately after incomplete token
```

```
                36 /w/
            # just after 1 good token and a space
                # rest of string forms legal query
                    37 /w / puketza@k6
                # string ends immediately after incomplete token
                    38 /w /
            # just after an otherwise legal query
                    39 /w puketza@k6/
            # just after an otherwise legal query and a space
                    40 /w puketza@k6 /
            # at 2nd-to-last possible position after an otherwise legal
            #  query (NL is in the last position)
                41 /w puketza@k6 ... /\n     #255 spaces between '6'
                                             #and '/'
        # (after NL):
            # just after an otherwise legal query
                42 /w puketza@k6\n/
            # just after an otherwise legal query and a space
                43 /w puketza@k6\n /
            # at last possible position after an otherwise legal query
                44 /w puketza@k6\n /         #255 spaces bet. NL and '/'


# Queries with at least one token-related problem:
  # Queries with missing tokens:
  #   (Note: a missing token in one query format sometimes creates
  #    a legal query in a different format. That is why some test
  #    cases appear to be missing below.)
    # no newline (i.e., missing NL token), each format:
        1
        2 /w
        3 /w puketza
        4 @k6
        5 puketza@k6
        6 /w @k6
        7 /w puketza@k6
    # 1 missing token, "/w username" format
        8  puketza              #missing 1st token (note
  # leading space)
        9 /wpuketza             #missing 2nd token (space)
        10 /w                   #missing 3rd token (space is
                                # last char)
    # 1 missing token, "@hostname" format
        11 k6                    #missing 1st token (@)
```

```
      12 k6@olympus              #missing 1st token (1st @ is
                                 # missing)
      13 @                       #missing 2nd token
  # 1 missing token, "username@hostname" format
      14 puketzak6               #missing 2nd token (@)
      15 puketzak6@olympus       #missing 2nd token (1st @ is missing)
      16 puketza@                #missing 3rd token (hostname)
  # 1 missing token, "/w @hostname" format
      17 @k6                     #missing 1st token (note leading
                                 #space)
      18 /w@k6                   #missing 2nd token (space)
      19 /w@k6@olympus           #missing 2nd token
      20 /w k6                   #missing 3rd token (@)
      21 /w k6@olympus           #missing 3rd token (1st @ is missing)
      22 /w @                    #missing 4th token (hostname)
  # 1 missing token, "/w username@hostname" format
      23 puketza@k6              #missing 1st token (note leading
                                 # space)
      24 /wpuketza@k6            #missing 2nd token (space)
      25 /wpuketza@k6@olympus    #missing 2nd token (space)
      26 /w puketzak6            #missing 4th token (@)
      27 /w puketzak6@olympus    #missing 4th token (1st @ is missing)
      28 /w puketza@             #missing 5th token (hostname)
# Queries with extra tokens:
  # (before the newline):
    # "empty string" format
      29 puketza                 #extra username token
    # "/w" format
      30 /w@                     #extra @ token
      31 /w/w                    #extra /w token
    # "/w username" format
      32 /w puketza/w            #extra /w token
      33 /w puketza@             #extra @ token
    # "@hostname" format
      34 @k6/w                   #extra /w token
      35 @k6@                    #extra @ token
    # "username@hostname" format
      36 puketza@k6/w            #extra /w token
      37 puketza@k6@             #extra @ token
    # "/w @hostname" format
      38 /w @k6/w                #extra /w token
      39 /w @k6@                 #extra @ token
    # "/w username@hostname" format
      40 /w puketza@k6/w         #extra /w token
```

```
    41 /w puketza@k6@         #extra @ token
  # (after the newline):
    Repeat all test cases related to an extra token
    before the newline, except move the extra token
    to just after the newline. (Cases 41-54).
# Queries with an incorrect token type in at least 1 position:
    # /w format:
        55 puketza           #username instead of /w
        56 @                 #@ instead of /w
        57                   #a space instead of /w
    # Note: Comments below indicate which token has wrong type.
    # /w user format:
        58 puketza puketza  #1st token
        59 /w@puketza       #2nd token
        60 /w @             #3rd token
        61 /w /w            #3rd token
    # @host format:
        62  k6              #1st token
        63 /wk6             #1st token
        64 @/w              #2nd token
        65 @@               #2nd token
    # username@host format:
        66 @@k6             #1st token
        67 puketza/wk6      #2nd token
        68 puketza@/w       #3rd token
        69 puketza@@        #3rd token
    # /w @host format:
        70 @ @k6            #1st token
        71 /w/w@k6          #2nd token
        72 /w /wk6          #3rd token
        73 /w @/w           #4th token
    # /w username@host format:
        74 @ puketza@k6     #1st token
        75 /w/wpuketza@k6   #2nd token
        76 /w @@k6          #3rd token
        77 /w puketza/wk6   #4th token
        78 /w puketza@/w    #5th token
```

Bibliography

[1] Ugen J. S. Antsilevich, Poul-Henning Kamp, Alex Nash, Archie Cobbs, and Luigi Rizzo. The FreeBSD "ipfw man page". [web page]; Feb. 16, 2000; http://www.freebsd.org/cgi/man.cgi?manpath=FreeBSD+4.0-current (Accessed Oct. 14, 2000).

[2] R. G. Bace, Division of Infosec Computer Science, Research and Technology, National Security Agency, private communication, May 1995.

[3] S. M. Bellovin. "Security Problems in the TCP/IP Protocol Suite". *ACM Computer Communication Review*, 19(2):32–48, April 1989.

[4] S. M. Bellovin. "There Be Dragons". In *Proc., Third USENIX UNIX Security Symposium*, pages 1–16, Baltimore, MD, September 1992.

[5] M. Bishop. "A Taxonomy of UNIX System and Network Vulnerabilities". Technical Report CSE-95-10, University of California, Davis, September 1995.

[6] K. A. Bradley. Detecting Disruptive Routers: A Distributed Network Monitoring Approach. Master's thesis, University of California, Davis, September 1997.

[7] 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, September/October 1998.

[8] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson. "Detecting Disruptive Routers: A Distributed Network Monitoring Approach". In *Proc. 1998 IEEE Symposium on Security and Privacy*, pages 115–124, Oakland, CA, May 1998.

[9] P. Brinch Hansen. "Reproducible Testing of Monitors". *Software–Practice and Experience*, 8(6):721–729, 1978.

[10] Sean Carroll. "Protect Your Desktop". *PC Magazine*, June 2000. http://www.zdnet.com/pcmag/stories/reviews/0,6755,2580291,00.html (Accessed Dec. 3, 2000).

[11] J. Case, M. Fedor, M. Schoffstall, and J. Davin. "A Simple Network Management Protocol (SNMP)", May 1990. RFC 1157.

[12] CERT Coordination Center. "CERT Advisory CA-99-04 Melissa Macro Virus," March 31, 1999.
http://www.cert.org/advisories/CA-1999-04.html
(Accessed Dec. 3, 2000).

[13] CERT Coordination Center. "CERT Advisory CA-2000-03 Continuing Compromises of DNS servers", April 26, 2000.
http://www.cert.org/advisories/CA-2000-03.html
(Accessed August 13, 2000).

[14] CERT Coordination Center. "CERT Advisory CA-1997-28 IP Denial-of-Service Attacks," May 26, 1998.
http://www.cert.org/advisories/CA-1997-28.html
(Accessed Dec. 3, 2000).

[15] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley, 1994.

[16] S. Cheung and K. N. Levitt. "Protecting Routing Infrastructures from Denial of Service Using Cooperative Intrusion Detection". In *Proc. New Security Paradigms Workshop*, Cumbria, UK, September 1997.

[17] M. Chung, N. Puketza, R. A. Olsson, and B. Mukherjee. "Simulating Concurrent Intrusions for Testing Intrusion Detection Systems: Parallelizing Intrusions". In *Proc. 18th National Information Systems Security Conference*, pages 173–183, Baltimore, MD, October 1995.

[18] D. E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture, Vol. 1, Third Edition.* Prentice Hall, 1995.

[19] Computer Incident Advisory Capability (CIAC). "H-12: IBM AIX(r) 'SYN Flood' and 'Ping o' Death' Vulnerabilities", December 10, 1996.
http://ciac.llnl.gov/ciac/bulletins/h-12.shtml
(Accessed August 13, 2000).

[20] Cookie Central. "Cookies," Copyright 1997-1998.
http://www.cookiecentral.com/cm002.htm
(Accessed November 8, 2000).

[21] Cookie Central. "The Dark Side of Cookies," Copyright 1997-1998.
http://www.cookiecentral.com/dsm.htm
(Accessed November 7, 2000).

[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[23] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade". In *Proc., DARPA Information Survivability Conference & Exposition (DISCEX).* IEEE Computer Society Press, January 2000.

[24] D. E. Denning. "An Intrusion-Detection Model". *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[25] D. Farmer and E. H. Spafford. "The COPS Security Checker System". In *Proc., Summer USENIX Conference*, pages 165–170, June 1990.

[26] D. Farmer and W. Venema. "Improving the Security of Your Site by Breaking Into It". Usenet posting, December 2, 1993.
http://www.alw.nih.gov/Security/Docs/admin-guide-to-cracking.101.html.

[27] S. Garfinkel. "50 Ways to Crash the Net". [web page, Hot Wired web site] Aug. 1997;
http://www.hotwired.com/synapse/feature/97/33/garfinkel0a_text.html
(Accessed 9 Feb. 2000).

[28] S. Garfinkel and G. Spafford. *Practical UNIX and Internet Security.* O'Reilly & Associates, Inc., second edition, 1996.

[29] L. D. Gary. Presentation, "Crime on the Internet" session, *17th National Computer Security Conference*, Baltimore, MD, Oct. 1994.

[30] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[31] L. T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber. "A Network Security Monitor". In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 296–304, Oakland, CA, May 1990.

[32] Hobbit. Software package "Readme" file: "Netcat 1.10". Package available from FreeBSD web site:
http://www.freebsd.org/cgi/ports.cgi?query=netcat&stype=all
(Accessed Dec. 3, 2000).

[33] Craig A. Huegen. "The Latest in Denial of Service Attacks: 'Smurfing' Description and Information to Minimize Effects", July 22, 1998.
http://saturn.waycom.com/smurf.htm
(Accessed August 13, 2000).

[34] J.R. Hughes, T. Aura, and M. Bishop. "Using Conservation of Flow as a Security Mechanism in Network Protocols". In *Proc., IEEE Symposium on Security and Privacy*, pages 132–141, Oakland, CA, May 2000.

[35] C. Huitema. *Routing in the Internet.* Prentice Hall, 1995.

[36] K. Ilgun, R. A. Kemmerer, and P. A. Porras. "State Transition Analysis: A Rule-Based Intrusion Detection Approach". *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[37] September 1981. Information Sciences Institute. Jon Postel, editor. RFC 793: "Transmission Control Protocol: DARPA Internet Program Protocol Specification".

[38] H. S. Javitz and A. Valdes. "The SRI IDES Statistical Anomaly Detector". In *Proc. IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1991.

[39] L. Joncheray. "A Simple Active Attack Against TCP". In *Proc. 5th USENIX UNIX Security Symposium*, pages 7–19, Salt Lake City, Utah, June 1995.

[40] S. Kent and R. Atkinson. Rfc 2401: "Security Architecture for the Internet Protocol", November 1998.

[41] S. Kumar and E. H. Spafford. "A Pattern Matching Model for Misuse Intrusion Detection". In *Proc. 17th National Information Systems Security Conference*, pages 11–21, Baltimore, MD, October 1994.

[42] S. Kumar and E. H. Spafford. "An Application of Pattern Matching in Intrusion Detection". Technical Report CSD-TR-94-013, Purdue University, June 17 1994.

[43] S. Kumar and E. H. Spafford. "A Software Architecture to Support Misuse Intrusion Detection". Technical Report CSD-TR-95-009, Purdue University, March 17 1995.

[44] C. E. Landwehr et al. "A Taxonomy of Computer Program Security Flaws". *ACM Computing Surveys*, 26(3):211–254, September 1994.

[45] T. J. LeBlanc and J. M. Mellor-Crummey. "Debugging Parallel Programs With Instant Replay". *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[46] David Legard and Stephen Lawson. "Love Letter worm rated most damaging ever," IDG News Service, May 5, 2000.
http://www.nwfusion.com/news/2000/0505lovebug2.html
(Accessed August 13, 2000).

[47] D. Libes. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs.* O'Reilly & Associates, Inc., 1994.

[48] T. F. Lunt. "Automated Audit Trail Analysis and Intrusion Detection: A Survey". In *Proc. 11th National Computer Security Conference*, pages 65–73, October 1988.

[49] T. F. Lunt et al. "A Real-Time Intrusion Detection Expert System(IDES)", May 1990. Interim Progress Report, Project 6784, SRI International.

[50] T. F. Lunt et al. "IDES: A Progress Report". In *Proc., Sixth Annual Computer Security Applications Conference*, Tucson, AZ, December 1990.

[51] S. A. Miller. Specification of Network Access Policy and Verification of Compliance Through Passive Monitoring. Master's thesis, University of California, Davis, 1999.

[52] John Moy. RFC 2178: "OSPF Version 2", July 1997.

[53] B. Mukherjee, L. T. Heberlein, and K. N. Levitt. "Network Intrusion Detection". *IEEE Network*, 8(3):26–41, May/June 1994.

[54] Biswanath Mukherjee. Unit II, Course Notes for Engineering Computer Science (ECS) 250: Local and Metropolitan Area Networks, University of California, Davis.

[55] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[56] P. G. Neumann and D. B. Parker. "A Summary of Computer Misuse Techniques". In *Proc., 12th National Computer Security Conference*, pages 396–407, Baltimore, MD, October 1989.

[57] Peter G. Neumann and Phillip A. Porras. "Experience with EMERALD to Date". In *Proc. 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, CA, April 1999.

[58] Patrick Oonk. "Promail trojan". [web page: "Bugtraq" mailing list archive, March 19, 1999. Maintained by SecurityFocus.com .]
http://securityfocus.com/ (Accessed August 13, 2000).

[59] T. J. Ostrand and M. J. Balcer. "The Category-Partition Method for Specifying and Generating Functional Tests". *Communications of the ACM*, 31(6), June 1988.

[60] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[61] R. Perlman. *Network Layer Protocols with Byzantine Robustness.* PhD thesis, Massachusetts Institute of Technology, October 1988.

[62] R. Perlman. *Interconnections: Bridges and Routers.* Addison-Wesley, 1992.

[63] N. Puketza, M. Chung, R. A. Olsson, and B. Mukherjee. "A Software Platform for Testing Intrusion Detection Systems". *IEEE Software*, 14(5):43–51, Sept./Oct. 1997.

[64] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. "Implementing a Generalized Tool for Network Monitoring". In *Proc. 11th Systems Administration Conference (LISA '97)*, San Diego, CA, October 1997.

[65] Martin Roesch. "Snort – Lightweight Intrusion Detection for Networks". In *Proc. 13th Systems Administration Conference (LISA '99)*, Seattle, WA, November 1999.

[66] Phillip Rogaway. "Bucket Hashing and its Application to Fast Message Authentication". *Journal of Cryptology*, 12(2):91–115, 1999.

[67] D. R. Safford, D. L. Schales, and D. K. Hess. "The TAMU Security Package: An Ongoing Response to Internet Intruders in an Academic Environment". In *Proc., Fourth USENIX UNIX Security Symposium*, pages 91–118, Santa Clara, CA, October 1993.

[68] C. Schuba et al. "Analysis of a Denial of Service Attack on TCP". In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 208–223, Oakland, CA, May 1997.

[69] Larry Seltzer. "Symantec Norton Internet Security 2000 2.0". *PC Magazine*, June 2000. http://www.zdnet.com/pcmag/stories/pipreviews/0,9836,254333,00.html (Accessed Dec. 3, 2000).

[70] S. E. Smaha. "Haystack: An Intrusion Detection System". In *Proc., IEEE Fourth Aerospace Computer Security Applications Conference*, Orlando, FL, December 1988.

[71] B. C. Smith, L. A. Rowe, and S. C. Yen. *Tcl Distributed Programming.* Tcl-DP Distribution Package, Computer Science Division, University of California, Berkeley, 1994.

[72] S.R. Snapp, B. Mukherjee, et al. "DIDS (Distributed Intrusion Detection System) — Motivation, Architecture, and an Early Prototype". In *Proc. 14th National Computer Security Conference*, pages 167–176, Washington, DC, Oct. 1991.

[73] B. Soh and T. Dillon. "Setting Optimal Intrusion-Detection Thresholds". *Computers and Security*, 14(7):621–31, 1995.

[74] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. "GrIDS – A Graph-Based Intrusion Detection System for Large Networks". In *Proc. 19th National Information Systems Security Conference*, pages 361–370, Baltimore, MD, October 1996.

[75] M. Steenstrup. *Routing in Communications Networks.* Prentice Hall, 1995.

[76] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley Publishing Co., 1994.

[77] Symantec Corp. "Virus Descriptions".
http://www.symantec.com/avcenter/virus.backgrounder.html
(Accessed August 13, 2000).

[78] Symantec Corp. "VBS.LoveLetter.A". [web page] May 9, 2000;
http://www.symantec.com/avcenter/venc/data/vbs.loveletter.a.html
(Accessed August 14, 2000).

[79] Michael A. Vatis. "Congressional Statement for the Record on Melissa Macro Virus," April 15, 1999.
http://www.fbi.gov/pressrm/congress/congress99/vatis1.htm
(Accessed August 13, 2000).

[80] G. Vigna and R. Kemmerer. "NetSTAT: A Network-Based Intrusion Detection System". *Journal of Computer Security*, 7(1), 1999.

[81] Elaine J. Weyuker and Bingchiang Jeng. "Analyzing Partition Testing Strategies". *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[82] K. Zhang. "A Methodology for Testing Intrusion Detection Systems". Master's thesis, University of California, Davis, May 1993.

[83] D. Zimmerman. RFC 1288: "The Finger User Information Protocol", December 1991.