1 INTRODUCTION

It was a dark and stormy night. Somewhere in the distance a dog howled. A shiny object caught Alice's eye. A diamond cufflink! Only one person in the household could afford diamond cufflinks! So it was the butler, after all! Alice had to warn Bob. But how could she get a message to him without alerting the butler? If she phoned Bob, the butler might listen on an extension. If she sent a carrier pigeon out the window with the message taped to its foot, how would Bob know it was Alice that was sending the message and not Trudy attempting to frame the butler because he spurned her advances?

That's what this book is about. Not much character development for Alice and Bob, we're afraid; nor do we really get to know the butler. But we do discuss how to communicate securely over an insecure medium.

What do we mean by communicating securely? Alice should be able to send a message to Bob that only Bob can understand, even though Alice can't avoid having others see what she sends. When Bob receives a message, he should be able to know for certain that it was Alice who sent the message, and that nobody tampered with the contents of the message in the time between when Alice launched the message and Bob received it.

What do we mean by an insecure medium? Well, in some dictionary or another, under the definition of "insecure medium" should be a picture of the Internet. The world is evolving towards interconnecting every computer, and people talk about connecting household appliances as well, all into some wonderful global internetwork. How wonderful! You'd be able to send electronic mail to anyone in the world. You'd also be able to control your nuclear power plant with simple commands sent across the network while you were vacationing in Fiji. Or sunny Libya. Or historic Iraq. Inside the network the world is scary. There are links that eavesdroppers can listen in on. Information needs to be forwarded through packet switches, and these switches can be reprogrammed to listen to or modify data in transit.

The situation might seem hopeless, but we may yet be saved by the magic of mathematics, and in particular cryptography, which can take a message and transform it into a bunch of numbers known as ciphertext. The ciphertext is unintelligible gibberish except to someone who knows the secret to reversing the transformation. Cryptography allows us to disguise our data so that eavesdroppers gain no information from listening to the information as transmitted. Cryptography also allows us to create an unforgeable message and detect if it has been modified in transit. One method of accomplishing this is with a **digital signature**, a number associated with a message and its sender that can be verified as authentic by others, but can only be generated by the sender. This should seem astonishing. How can there be a number which you can verify but not generate? A person's handwritten signature can (more or less) only be generated by that person, though it can be verified by others. But it would seem as if a number shouldn't be hard to generate, especially if it can be verified. Theoretically, you could generate someone's signature by trying lots of numbers and testing each one until one passed the verification test. But with the size of the numbers used, it would take too much compute time (for instance, several universe lifetimes) to generate the signature that way. So a digital signature has the same property as a handwritten signature, in that it can only be generated by one person. But a digital signature does more than a handwritten signature. Since the digital signature depends on the contents of the message, if someone alters the message the signature will no longer be correct and the tampering will be detected. This will all become clear if you read Chapter 2 *Introduction to Cryptography*.

Cryptography is a major theme in this book, not because cryptography is intrinsically interesting (which it is), but because many of the security features people want in a computer network can best be provided through cryptography.

1.1 ROADMAP TO THE BOOK

After this introductory chapter, there are five main sections in the book:

- **Part 1 CRYPTOGRAPHY** Chapter 2 *Introduction to Cryptography* is the only part of the cryptography section of the book essential for understanding the rest of the book, since it explains the generic properties of secret key, message digest, and public key algorithms, and how each is used. We've tried our best to make the descriptions of the actual cryptographic algorithms nonthreatening yet thorough, and to give intuition into why they work. It's intended to be readable by anyone, not just graduate students in mathematics. Never once do we use the term *lemma*. We do hope you read Chapter 3 *Secret Key Cryptography*, Chapter 4 *Modes of Operation*, Chapter 5 *Hashes and Message Digests*, and Chapter 6 *Public Key Algorithms* which give the details of the popular standards, but it's also OK to skip them and save them for later, or just for reference. Chapter 7 *Number Theory* and Chapter 8 *Math with AES and Elliptic Curves* gives a deeper treatment of the mathematics behind the cryptography. Reading them is not necessary for understanding the rest of the book.
- **Part 2 AUTHENTICATION** Chapter 9 *Overview of Authentication Systems* introduces the general issues involved in proving your identity across a network. Chapter 10 *Authentication of People* deals with the special circumstances when the device proving its identity is a

human being. Chapter 11 *Security Handshake Pitfalls* deals with the details of authentication handshakes. There are many security flaws that keep getting designed into protocols. This chapter attempts to describe variations of authentication handshakes and their relative security and performance strengths. We end the chapter with a checklist of security attacks so that someone designing a protocol can specifically check their protocol for these flaws.

- **Part 3 STANDARDS** This portion of the book describes the standards: Kerberos versions 4 and 5, certificate and PKI standards, IPsec, and SSL. We hope that our descriptions will be much more readable than the standards themselves. And aside from just describing the standards, we give intuition behind the various choices, and criticisms where they are overly complex or have flaws. We hope that our commentary will make the descriptions more interesting and provide a deeper understanding of the design decisions. Our descriptions are not meant to, and cannot, replace reading the standards themselves, since the standards are subject to change. But we hope that after reading our description, it will be much easier to understand the standards.
- **Part 4 ELECTRONIC MAIL** Chapter 20 *Electronic Mail Security* describes the various types of security features one might want, and how they might be provided. Chapter 21 *PEM & S/MIME* and Chapter 22 *PGP (Pretty Good Privacy)* describe the specifics of PEM, S/MIME, and PGP.
- **Part 5 LEFTOVERS** Chapter 23 *Firewalls* talks about what firewalls are, what problems they solve, and what problems they do not solve. Chapter 24 *More Security Systems*, describes a variety of security systems, including Novell NetWare (Versions 3 and 4), Lotus Notes, DCE, KryptoKnight/NetSP, Clipper, SNMP, DASS/SPX, Microsoft (LAN Manager and Windows NT), and sabotage-proof routing protocols. Chapter 25 *Web Issues* talks about the protocols involved in web surfing: URLs, HTTP, HTML, cookies, etc., and the security issues these raise. We close with Chapter 26 *Folklore*, which describes the reasoning behind some of the advice you will hear from cryptographers.

1.2 WHAT TYPE OF BOOK IS THIS?

We believe the reason most computer science is hard to understand is because of jargon and irrelevant details. When people work with something long enough they invent their own language, come up with some meta-architectural framework or other, and forget that the rest of the world doesn't talk or think that way. We intend this book to be reader-friendly. We try to extract the concepts and ignore the meta-architectural framework, since whatever a meta-architectural framework is, it's irrelevant to what something does and how it works.

We believe someone who is a relative novice to the field ought to be able to read this book. But readability doesn't mean lack of technical depth. We try to go beyond the information one might find in specifications. The goal is not just to describe exactly how the various standards and de facto standards work, but to explain why they are the way they are, why some protocols designed for similar purposes are different, and the implications of the design decisions. Sometimes engineering tradeoffs were made. Sometimes the designers could have made better choices (they are human after all), in which case we explain how the protocol could have been better. This analysis should make it easier to understand the current protocols, and aid in design of future protocols.

The primary audience for this book is engineers, especially those who might need to evaluate the security of, or add security features to, a distributed system; but the book is also intended to be usable as a textbook, either on the advanced undergraduate or graduate level. Most of the chapters have homework problems at the end.

Not all the chapters will be of interest to all readers. In some cases we describe and critique a standard in great detail. These chapters might not be of interest to students or people trying to get a conceptual understanding of the field. But in many cases the standards are written fairly unintelligibly. People who need to understand the standard, perhaps to implement it, or maybe even to use it, need to have a place where it is described in a readable way (and we strive for readability), but also a place in which mistakes in the standard are pointed out as such. It's very difficult to understand why, for instance, two fields are included which both give the same information. Sometimes it is because the designers of the protocol made a mistake. Once something like that is pointed out as a simple mistake, it's much easier to understand the specification. We hope that reading the descriptions in the book will make the specifications more intelligible.

1.3 TERMINOLOGY

Computer science is filled with ill-defined terminology used by different authors in conflicting ways, often by the same author in conflicting ways. We apologize in advance for probably being guilty sometimes ourselves. Some people take terminology very seriously, and once they start to use a certain word in a certain way, are extremely offended if the rest of the world does not follow.

> *When I use a word, it means just what I choose it to mean—neither more nor less.*
>
> —Humpty Dumpty (in *Through the Looking Glass*)

Some terminology we feel fairly strongly about. We do *not* use the term *hacker* to describe the vandals that break into computer systems. These criminals call themselves hackers, and that is how they got the name. But they do not deserve the name. True hackers are master programmers, incorruptibly honest, unmotivated by money, and careful not to harm anyone. The criminals termed "hackers" are not brilliant and accomplished. It is really too bad that they not only steal money, people's time, and worse, but they've also stolen a beautiful word that had been used to describe some remarkable and wonderful people. We instead use words like *intruder*, *bad guy*, and *impostor*. When we need a name for a bad guy, we usually choose *Trudy* (since it sounds like *intruder*).

We grappled with the terms *secret key* and *public key* cryptography. Often in the security literature the terms *symmetric* and *asymmetric* are used instead of *secret* and *public*. We found the terms *symmetric* and *asymmetric* intimidating and sometimes confusing, so opted instead for *secret key* and *public key*. We occasionally regretted our decision to avoid the words *symmetric* and *asymmetric* when we found ourselves writing things like *secret key based interchange keys* rather than *symmetric interchange keys*.

We use the term *privacy* when referring to the desire to keep communication from being seen by anyone other than the intended recipients. Some people in the security community avoid the term *privacy* because they feel its meaning has been corrupted to mean *the right to know*, because in some countries there are laws known as *privacy laws* which state that citizens have the right to see records kept about themselves. *Privacy* also tends to be used when referring to keeping personal information about people from being collected and misused. The security community also avoids the use of the word *secrecy*, because *secret* has special meaning within the military context, and they feel it would be confusing to talk about the secrecy of a message that was not actually labeled top secret or secret. The term most commonly used in the security community for keeping communication from being seen is *confidentiality*. We find that strange because *confidential*, like *secret*, is a security label, and the security community should have scorned use of *confidential*, too. In the first edition, we chose not to use *confidentiality* because we felt it had too many syllables, and saw no reason not to use *privacy*. For the second edition we reconsidered this decision, and were about to change all use of *privacy* to *confidentiality* until one of us pointed out we'd have to change the title of the book to something like *Network Security: Confidential Communication in a Non-Confidential World*, at which point we decided to stick with *privacy*.

> Speaker: *Isn't it terrifying that on the Internet we have no privacy?*
> Heckler$_1$: *You mean* confidentiality. *Get your terms straight.*
> Heckler$_2$: *Why do security types insist on inventing their own language?*
> Heckler$_3$: *It's a denial-of-service attack.*
>
> —Overheard at recent gathering of security types

We often refer to things involved in a conversation by name, for instance, *Alice* and *Bob*, whether the things are people or computers. This is a convenient way of making things unambiguous with relatively few words, since the pronoun *she* can be used for Alice and *he* can be used for Bob. It also avoids lengthy inter- (and even intra-) author arguments about whether to use the politically incorrect *he*, a confusing *she*, an awkward *he/she* or *(s)he*, an ungrammatical *they*, an impersonal *it*, or an incredibly awkward rewriting to avoid the problem. We remain slightly worried that people will assume when we've named things with human names that we are referring to people. Assume Alice, Bob, and the rest of the gang may be computers unless we specifically say something like *the user Alice*, in which case we're talking about a human.

> *With a name like yours, you might be any shape, almost.*
>
> —Humpty Dumpty to Alice (in *Through the Looking Glass)*

Occasionally, one of the three of us authors will want to make a personal comment. In that case we use *I* or *me* with a subscript. When it's a comment that we all agree with, or that we managed to slip past me_3 (the rest of us are wimpier), we use the term *we*.

1.4 NOTATION

We use the symbol ⊕ (pronounced *ex-or*) for the bitwise-exclusive-or operation. We use the symbol | for concatenation. We denote secret key encryption with curly brackets preceded by the key with which something was encrypted, as in $K\{\textit{message}\}$, which means *message* is secret key encrypted with *K*. Public key encryption we denote with curly braces, and the name of the owner of the public key subscripting the close brace, as in $\{\textit{message}\}_{\text{Bob}}$. Signing (which means using the private key), we denote with square brackets, with the name of the owner of the key subscripting the close bracket, as in $[\textit{message}]_{\text{Bob}}$.

Table of Notation

⊕	bitwise exclusive or (pronounced *ex-or*)
|	concatenation (pronounced *concatenated with*)
$K\{\textit{message}\}$	*message* encrypted with secret key *K*
$\{\textit{message}\}_{\text{Bob}}$	*message* encrypted with Bob's public key
$[\textit{message}]_{\text{Bob}}$	*message* signed with Bob's private key

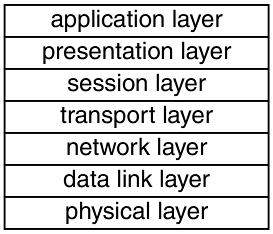
1.5 PRIMER ON NETWORKING

You have to know something about computer networks to understand computer network security, so we're including this primer. For a more detailed understanding, we recommend PERL99, TANE96, COME00, STEV94, KURO00.

Networks today need to be very easy to use and configure. Networks are no longer an expensive educational toy for researchers, but instead are being used by real people. Most sites with networks will not be able to hire a full-time person with networking expertise to start and keep the network running.

1.5.1 OSI Reference Model

Somehow, a book about computer networks would seem incomplete without a picture of the OSI (Open Systems Interconnection) Reference Model, so here it is.



application layer
presentation layer
session layer
transport layer
network layer
data link layer
physical layer

Figure 1-1. OSI Reference Model

The OSI Reference Model is useful because it gives some commonly used terminology, though it might mislead you into thinking that there is only one way to construct a network. The reference model was designed by an organization known as the International Standards Organization (ISO). The ISO decided it would be a good idea to standardize computer networking. Since that was too big a task for a single committee, they decided to subdivide the problem among several committees. They somewhat arbitrarily chose seven, each responsible for one layer. The basic idea is that each layer uses the services of the layer below, adds functionality, and provides a service to the layer above. When you start looking at real networks, they seldom neatly fit into the seven-layer model, but for basic understanding of networking, the OSI Reference Model is a good place to start.

1. **physical layer**. This layer delivers an unstructured stream of bits across a link of some sort.
2. **data link layer**. This layer delivers a piece of information across a single link. It organizes the physical layer's bits into packets and controls who on a shared link gets each packet.

3. **network layer**. This layer computes paths across an interconnected mesh of links and packet switches, and forwards packets over multiple links from source to destination.
4. **transport layer**. This layer establishes a reliable communication stream between a pair of systems across a network by putting sequence numbers in packets, holding packets at the destination until they can be delivered in order, and retransmitting lost packets.
5. **session layer**. The OSI session layer adds extra functions to the reliable pair-wise communication provided by the transport layer. Most network architectures do not have or need the functionality in this layer, and it is not of concern to security, so for the purposes of this book we can ignore it.
6. **presentation layer**. This layer encodes application data into a canonical (system-independent) format and decodes it into a system-dependent format at the receiving end.
7. **application layer**. This is where the applications that use the network, such as web surfing, file transfer, and electronic mail, reside.

A layer communicates with the equivalent layer in a different node. In order to get data to a peer layer, though, the layer at the transmitting node gives the data to the layer below it (on the same node), which adds a header containing additional information if necessary, and that layer in turn gives it to the layer below. As the packet is received by the destination node, each layer reads and strips off its own header, so that the packet received by layer *n* looks to that layer just like it did when it was sent down to layer *n*−1 for transmission.

This seven-layer model is the basis for a lot of the terminology in networking, and a good first step towards understanding networks, but today's network protocols do not neatly fit this model. Throughout the book we sometimes use the OSI terminology by discussing things such as encryption at layer 2 vs. layer 3 vs. layer 4, or use the terms *data link layer*, or *transport layer.*

1.5.2 IP, UDP, and TCP

Today the most common protocols are the ones standardized by the IETF (Internet Engineering Task Force). All the IETF documents are on-line and freely available from the web site www.ietf.org. The protocols are specified in documents known as RFCs. (RFC is an abbreviation for "Request for Comments", but the time to comment is when the documents are in the more preliminary "internet draft" stage. Nobody wants to hear your comments on RFCs.)

The IETF's protocol suite is usually referred to as the "TCP/IP suite", after the most common layer 3 (IP) and layer 4 (TCP) protocols at the time the suite was being nicknamed. IP (Internet Protocol), the layer 3 protocol, is defined in RFC 791. Its job is to deliver data across a network. To get a letter mailed with the postal service, you put it in an envelope that specifies the source and destination address. Similarly, the IP layer adds an envelope (a header) to the data that specifies the source and destination addresses.

But the IP address only specifies the destination machine. There might be multiple processes at the destination machine all communicating across the network, so it's necessary to also specify which process should receive the data. This is similar to putting an apartment number on the envelope in addition to the street address. IP doesn't identify the processes, but instead has a 1-octet field that specifies which protocol should receive the packet, and the rest of the information necessary to identify the destination process is contained in the layer 4 header, in the PORT fields.

The two most important layer 4 protocols in the IETF suite are TCP (Transmission Control Protocol, defined in RFC 793) and UDP (User Datagram Protocol, defined in RFC 768). TCP sends an unlimited size stream of data, reliably (either all data is delivered to the other end without loss, duplication, or misordering, or the connection is terminated). UDP sends limited-sized individual chunks, with best-effort service. Both TCP and UDP have fields for SOURCE PORT and DESTINATION PORT, which specify the process to whom the data belongs. TCP additionally has sequence numbers and acknowledgments to ensure the data arrives reliably.

Some port numbers are "well-known", i.e., permanently assigned to a particular service, whereas others are dynamically assigned. Being able to contact someone at a well-known port makes it easy to establish communication. In contrast, if Alice and Bob were going to attempt to communicate by going to public telephones wherever they happened to be, they'd never be able to communicate, since neither one would know what number to call. But if one of them were listening at a well-known telephone number, then the other could call from anywhere. This is very similar to the use of well-known ports.

To communicate with a particular service, say the telnet service, at some machine at IP address x, you'd know that telnet uses TCP, and is always assigned to port 23. So in the IP header, you'd specify x as the destination address, and 6 (which means TCP) as the protocol type. In the TCP header, you'd specify port 23 as the destination port. Your process would be at a dynamically assigned port, but the recipient process at node x would know which port to reply to by copying the port from the source port in the received TCP header.

This will all become much more relevant when we discuss firewalls in Chapter 23 *Firewalls*, and how they can distinguish telnet packets (which firewall administrators would usually like to block) from, say, email packets (which firewall administrators would usually like to allow).

1.5.3 Directory Service

Having a telephone line into your house means you can access any phone in the world, if you know the telephone number. The same thing is true, more or less, in computer networks. If you know the network layer address of a node on the network, you should be able to communicate with that node. (This isn't always true because of security gateways, which we'll discuss in Chapter 23 *Firewalls*.)

But how do you find out another node's network layer address? Network layer addresses are not the kind of things that people will be able to remember, or type. People instead will want to access something using a name such as File-Server-3.

This is a similar problem to finding someone's telephone number. Typically you start out by knowing the name of the person or service you want to talk to, and then look the name up in a telephone book. In a computer network there is a service which stores information about a name, including its network layer address. Anything that needs to be found is listed in the service. Anything that needs to find something searches the service.

We call such a service a **directory**, though some people like to reserve the term "directory" for something in which you search based on an attribute (e.g., "find all the people who live on Main Street") rather than look up something based on knowing its name. Those people would call a simple service in which you look up information (rather than do complex searches) a **naming** service. We see no reason to make that distinction. It might be nice to search for all items that match a certain attribute, but usually the name will be known, and the attributes of that name will be fetched.

Rather than keeping all names in one directory, the directory service is typically structured as a tree of directories. Usually a name is hierarchical, so that the directory in which the name can be found is obvious from the name. For example, an Internet name looks like radia@east.sun.com. The top level consists of pointers to the directories com for commercial enterprises, edu for educational institutions, gov for U.S. government, and various country names. Under com, there are various company names.

Having multiple directories rather than keeping all names in one directory serves two purposes. One is to prevent the directory from getting unreasonably large. The other reason is to reduce name collisions (more than one object with the same name). For instance, when you're looking up a telephone number for your friend John Smith, it's bad enough trying to figure out which John Smith is the one you want if you know which town he lives in and the telephone company has separate directories for each town, but imagine if the telephone company didn't have separate books for each town and simply had a list of names and telephone numbers!

Ideally, with a hierarchy of directories, name collisions could be prevented. Once a company hired one Radia Perlman, they just wouldn't hire another. I_2 think that's reasonable, but someone with a name like John Smith might start having problems finding a company that could hire him.

> *Now why did you name your baby* John*? Every Tom, Dick, and Harry is named* John.
>
> —Sam Goldwyn

For electronic mail addresses, conflicts must be prevented. Typically, companies let the first John Smith use the name John@companyname for his email address, and then perhaps the next one will be Smith@companyname, and the next one JSmith@companyname, and the next one has to start using middle initials. But for directories of names, there is usually no way to avoid name collisions within a directory. In other words, both John Smiths will use the same name within the company. Then, just like with a telephone book and multiple John Smiths, you have to do the best you can to figure out which one you want based on various attributes (such as in the telephone directory, using the street address). And just like in "real life," there will be lots of confusion where one John Smith gets messages intended for a different John Smith.

The directory service is very important to security. It is assumed to be widely available and convenient to access—otherwise large-scale networking really is too inconvenient to be practical. The directory service is a convenient place to put information, such as a user's public cryptographic key. But the directory service, although convenient, is not likely to be very secure. An intruder might tamper with the information. The magic of cryptography will help us detect such tampering so that it will not be necessary to physically secure all locations that store directory service information. If the information is tampered with, good guys will detect this. It might prevent good guys from accessing the network, since they won't be able to find information they can trust, but it will not allow bad guys unauthorized access.

1.5.4 Replicated Services

Sometimes it is convenient to have two or more computers performing the same function. One reason is performance. A single server might become overloaded, or might not be sufficiently close to all users on a large network. Another reason is availability. If the service is replicated, it does not matter if some of the replicas are down or unavailable. When someone wants to access the service provided, it doesn't matter which of the computers they reach. Often the user can't even tell whether there's a single copy of the service or there are replicas.

What are the security issues with a replicated service? You'd want the user to have the same authentication information regardless of which replica was authenticating the user. If authentication information is stored at each replica, then coordinating the databases, for example after a change password command, can be tricky. And if the identical exchange will work with any of the replicas, then having an eavesdropper repeat the authentication handshake with a different replica might be a security problem.

1.5.5 Packet Switching

A really naive assumption would be that if people wanted computer A to talk to computer B, they'd string a wire between A and B. This doesn't work if networks get large, either in number of nodes (n^2 wires) or physical distance (it takes a lot of wire to connect each of 10000 nodes in North America with each of 10000 nodes in Asia). So in a network, messages do not go directly from sender to recipient, but rather have to be forwarded by various computers along the way. These message forwarders are referred to as packet switches, routers, gateways, bridges, and probably lots of other names as well.

A message is generally broken into smaller chunks as it is sent through the network. There are various reasons for this.

- Messages from various sources can be interleaved on the same link. You wouldn't want your message to have to wait until someone else finished sending a huge message, so messages are sent a small chunk at a time. If the link is in the process of sending the huge message when your little single-chunk message arrives, your message only has to wait until the link finishes sending a chunk of the large message.
- Error recovery is done on the chunk. If you find out that one little chunk got mangled in transmission, only that chunk needs to be retransmitted.
- Buffer management in the routers is simpler if the size of packets has a reasonable upper limit.

1.5.6 Network Components

The network is a collection of packet switches (usually called routers) and links. A link can either be a wire between two computers or a multi-access link such as a LAN (local area network). A multi-access link has interesting security implications. Whatever is transmitted on the link can be seen by all the other nodes on that link. Multi-access links with this property include Ethernet (also known as CSMA/CD), token rings, and packet radio networks.

Connected to the backbone of the network are various types of nodes. A common categorization of the nodes is into *clients*, which are workstations that allow humans to access the resources on the network, and *servers*, which are typically dedicated machines that provide services such as file storage and printing. It should be possible to deploy a new service and have users be able to conveniently find the service. Users should be able to access the network from various locations, such as a public workstation a company makes available for visitors. If a person has a dedicated workstation located in one location, such as an office, it should be possible with a minimum of configuration for the user to plug the workstation into the network.

Historically, another method for users to access a network is through a *dumb terminal*. A dumb terminal is not a general-purpose computer and does not have the compute power to do cryptographic operations. Usually a dumb terminal hooks directly into a host machine, or into a *terminal server* which relays the terminal's keystrokes via a network protocol across the network to the host machine (the machine the user logs into). Very few dumb terminals remain today, but their legacy lives on in the form of software-based terminal emulators implemented in most PCs and workstations. Even though these devices are capable of complex calculations, for backward compatibility, they don't do them.

1.5.7 Destinations: Ultimate and Next-Hop

A network is something to which multiple systems can attach. We draw it as a cloud since, from the point of view of the systems connected to it, exactly what goes on inside is not relevant. If two systems are on the same cloud, one can send a message to the other by attaching a header that contains a source address and a destination address, much like putting a letter into an envelope for delivery by the postal service.

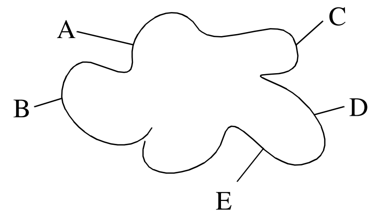


Figure 1-2. A Network

But how do you connect to the network? With a point-to-point link to a packet switch inside the network, things are reasonably simple. If A wants to send a message to B, A will put A as source address and B as destination address and send the message on the point-to-point link. But what if A is connected on a LAN? In that case, in order to transmit the packet through the network, A has to specify which of its neighbors should receive the message. For example:

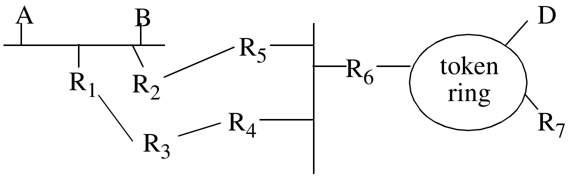


Figure 1-3. Network Connections

If A wants to send a message to D it has to know (somehow—if you care how, you can read my_2 book [PERL99]) that the appropriate neighbor for forwarding the packet is R_2. So when A transmits the message there are two destinations: R_2 as the next recipient and D as the ultimate recipient. A reasonably simple way of thinking about this is that the data link layer worries about transmission across a single link. The data link header has a source address and a destination address which indicate the transmitter on that link and the receiver on that link. The network layer worries about transmission across a multi-hop network. It has a header that carries the original source and ultimate destination. The data link header is removed each time a message is received, and a new data link header is tacked onto the message when it is forwarded to the next hop.

When A transmits the packet, the network header has source A, destination D. The data link header has source A, destination R_2. R_2 forwards the packet to R_5. Since R_2 is connected to R_5 with a point-to-point link, the data link header will not have addresses. But when R_5 forwards the packet to R_6 across the LAN, the network layer header will (still) be source A, destination D. The data link header will be source R_5, destination R_6. When R_6 forwards it (across the token ring LAN) the network header is still the same, and the data link header has source R_6, destination D.

Most likely A's data link address will look different from its network layer address, so it's a bit sloppy to say source A in both the data link header and network header. But this is all irrelevant to security. Fascinating in its own right, but irrelevant to this book.

The network layer header can be thought of as an envelope for the message. The data link header is an outer envelope. We've described the case of two envelopes—a network header inside a data link header. The world can be even more complicated than this. In fact, the "data link layer" might be a multi-hop network with multi-hop networks inside it as well. So a message might wind up with several envelopes. Again this is fascinating stuff but irrelevant to this book.

1.5.8 Address Structure

What do addresses look like? In terms of security, the main issue is how difficult it is to forge a source address, and how easy it is to arrange for the network to deliver packets to you when they are addressed to someone other than you. For instance, think of a letter as having a source address (the return address, it's called in paper mail) and a destination address. It's easy to send a letter to anyone and put President, White House, USA as the source address. It's harder to arrange to receive mail sent to President, White House, USA if you are not the U.S. President, especially if you don't live in the White House, and most likely more difficult the further you live from the address you'd like to impersonate. Network addresses are usually hierarchical, just like a postal address. If we think of the address as specifying country/state/city/person, then in general it will be easier to arrange to receive someone else's messages if you reside in the same city (for instance by bribing a postal employee), and most difficult if they're in a different country.

Forging source addresses is easy in most network layers today. Routers can be built more defensively and do a sanity check on the source address, based on where they receive the packet from. After some highly publicized denial of service attacks, where vandals overwhelmed victim sites with nuisance traffic, many routers are now deployed with this feature of checking source addresses and discarding traffic received from an unexpected direction. It's not a perfect solution, though. As typically implemented, it requires extra configuration (so the routers will know what source addresses to expect from which directions), somewhat violates my_2 philosophy (as a layer 3 specialist) that routers should be self-configuring and adapt to topological changes, and slows down the router because it has to make an extra check when forwarding a packet.

1.6 ACTIVE VS. PASSIVE ATTACKS

A passive attack is one in which the intruder eavesdrops but does not modify the message stream in any way. An active attack is one in which the intruder may transmit messages, replay old messages, modify messages in transit, or delete selected messages from the wire. A typical active attack is one in which an intruder impersonates one end of the conversation, or acts as a man-in-the-middle (see §6.4.1 *The Bucket Brigade/Man-in-the-Middle Attack*).

1.7 LAYERS AND CRYPTOGRAPHY

Encryption and integrity protection are sometimes done on the original message or on each chunk of the message, and if on each chunk, it might be done end-to-end or hop-by-hop. There are interesting tradeoffs and implications of these choices. If done on the original message, it can be protected while being stored, and the infrastructure does not need to even know whether the data it is moving is cryptographically protected. This means that the location where the cryptographically protected message is kept, and the infrastructure for transmitting the message, need not be trusted.

Encryption hop-by-hop can foil traffic analysis, i.e., it hides from eavesdroppers the information about which parties are communicating. Thus it is useful even if encryption is being done at other layers. If done hop-by-hop, the packet switches must be trusted, because by definition of hop-by-hop, the packet switches will see the plaintext.

If done end-to-end as the data is being transmitted, if individual chunks are separately encrypted and integrity protected, then the data that arrives intact can be used, whereas if there's only a single integrity check for the entire message, then any corruption or loss will require retransmitting the entire thing, since (by definition of cryptographically protecting the data as a whole instead of individual chunks) there will be no way to know where the loss/corruption occurred.

1.8 AUTHORIZATION

Network security basically attempts to answer two questions: "who are you?" and "should you be doing that?" **Authentication** proves who you are. **Authorization** defines what you're allowed to do. Typically the way a server decides whether someone should have access to a resource is by first authenticating the user, and then consulting a database associated with the resource that indicates who is allowed to do what with that resource. For instance, the database associated with a file might say that Alice can read it and Bob and Carol can both read and write it. This database is often referred to as an **ACL (access control list**).

Another model of authorization is known as the **capability model**. Instead of listing, with each resource, the set of authorized users and their rights (e.g., read, write, execute), you would have a database that listed, for each user, everything she was allowed to do.

If there were only a single resource, then the ACL model and the capability model would be basically the same, since in both cases there would be a database that lists all the authorized users and what rights each has to that resource. But in a world in which there are many resources, not all under control of one organization, it would be difficult to have a central database listing what each user was allowed to do (for instance, all the files that user is allowed to read), and it would have scaling problems if there were many resources each user was allowed to access.

Some people worry that ACLs don't scale well if there are many users allowed access to each resource. But the concept of a **group** answers that concern. A very basic form of group implemented in some systems is that each user is a member of one group, and someone with special privileges assigns users to groups. There is a special group known as "world", which includes everyone. Alice would be allowed to read a file if her name was listed on the ACL with read access, or if her group was listed on the ACL with read access, or if "world" was given read access.

Extensions to the idea of groups that might be useful:

- allow a user to be in multiple groups (researchers, security experts, U.S. citizens)
- allow anyone (not just someone with special privileges) to create a group. Allow anyone to name that group on an ACL they are authorized to administer.
- allow a group for which the user explicitly invokes his membership. This type of group is known as a **role**. The main difference between what people think of as a role and what people think of as a group is that the user always has all the rights of all the groups he is a member of, but only has the rights of the role he has explicitly invoked. Some people would claim that if the user is allowed to assert multiple roles, he can have only one of them active at any time.

We discuss ways of implementing very flexible notions of groups in §15.8.3 *Groups*.

1.9 TEMPEST

One security concern is having intruders tap into a wire, giving them the ability to eavesdrop and possibly modify or inject messages. Another security concern is electronic emanation, whereby through the magic of physics, the movement of electrons can be measured from a surprising distance away. This means that intruders can sometimes eavesdrop without even needing to physically access the link. The U.S. military Tempest program measures how far away an intruder must be before eavesdropping is impossible. That distance is known as the device's **control zone**. The control zone is the region that must be physically guarded to keep out intruders that might be attempting to eavesdrop. A well-shielded device will have a smaller control zone. I_1 remember being told in 1979 of a tape drive that had a control zone over two miles. Unfortunately, most control zone information is classified, and I_2 couldn't get me_1 to be very specific about them, other than that they're usually expressed in metric. Since it is necessary to keep intruders away from the control zone, it's certainly better to have something with a control zone on the order of a few inches rather than a few miles (oh yeah, kilometers).

> *CIA eavesdroppers could not intercept the radio transmissions used by Somali warlord Mohammed Farah Aidid; his radios, intelligence officials explained, were too "low tech."*
> —Douglas Waller & Evan Thomas, *Newsweek*, October 10, 1994, page 32

1.10 KEY ESCROW FOR LAW ENFORCEMENT

Law enforcement would like to preserve its ability to wiretap otherwise secure communication. (Also, sometimes companies want to be able to read all data of their employees, either to enforce company policies, or to ensure data is not lost when an employee forgets a password or leaves the company.)

In order for the government to ensure it can always wiretap, it must prevent use of encryption, break the codes used for encryption (as it did in a military context during World War II), or somehow learn everyone's cryptographic keys. The Clipper proposal was proposed in the mid-90's and attempted the third option. It allows the government to reconstruct your key (only upon court order and with legitimate cause of course). This is made possible through the use of a device known as the Clipper chip. A lot about Clipper was classified by the government as secret (and classified by a lot of other people as evil). We describe the basic technical design of Clipper in §24.9 *Clipper*. Although the Clipper proposal appears to have been a failure, and the government appears to have for the moment at least given up on attempting to control cryptography, the Clipper design was fascinating, and is worth learning about. The simple concept is that encryption is done with a special chip (the Clipper chip). Each chip manufactured has a unique key, and the government keeps a record of the serial number/encryption key correspondence of every chip manufactured. Because not all people have complete trust in the government, rather than keeping the key in one place, each key is broken into two quantities which must be ⊕'d in order to obtain the actual key. Each piece is completely useless without the other. Since each piece is kept with a separate government agency, it would require two U.S. government agencies to cooperate in order to cheat and obtain the key for your Clipper chip without a valid court order. The government assures us, and evidence of past experience supports its claim, that cooperation between U.S. government agencies is unlikely.

The Clipper proposal was always controversial, starting with its name (which violated someone's trademark on something unrelated). Why would anyone use Clipper when alternative methods should be cheaper and more secure? The reason alternatives would be cheaper is that enforcing the ability of the U.S. government to wiretap adds a lot of complexity over a design that simply encrypts the data. Proponents of Clipper gave several answers to this question:

- The government would buy a lot of Clipper chips, bringing the cost down because of volume production, so Clipper would wind up being the most cost-effective solution.
- Encryption technology is only useful if both parties have compatible equipment. Since the U.S. government would use Clipper, to talk securely to the U.S. government, you would have to use Clipper. So any other mechanism would have to be implemented *in addition* to Clipper.
- Again, since encryption technology is only useful if both parties have compatible equipment, if Clipper took over enough market share, it would essentially own the market (just like VHS, a technically inferior standard supposedly, beat out Beta in the VCR marketplace). Since Clipper would be one of the earliest standards, it might take over the marketplace before any other standards have an opportunity to become entrenched. The argument was that most people wouldn't care that Clipper enables wiretapping, because they'll assume they have nothing to fear from the U.S. government wiretapping them.
- The government claimed that the cryptographic algorithm in Clipper was stronger than you could get from a commercial source.

Civil libertarians feared Clipper was a first step towards outlawing untappable cryptography. Clipper proponents say it was not. It's true that outlawing alternatives was not part of the Clipper proposal. However, there have been independent efforts to outlaw cryptography. Those efforts have been thwarted in part with the argument that industry needs security. But if Clipper were deployed, that argument would have gone away.

Clipper was designed for telephones, fax, and other low-speed applications, and in some sense is not relevant to computer networking. Many people regard it, however, as a first step and a model for taking the same approach for computer networks.

The Clipper proposal was a commercial failure, and export controls are currently relaxed. However, the technical aspects of such designs are fascinating, laws can change at any time, and export controls have created other fascinating and arcane designs that we will describe throughout the book, for instance, §19.13 *Exportability*.

1.11 KEY ESCROW FOR CARELESS USERS

It is prudent to keep your key in a safe place so that when you misplace your own key you can retrieve a copy of the key rather than conceding that all your encrypted files are irretrievably lost. It would be a security risk to have all users' keys stored unencrypted somewhere. The database of keys could be stored encrypted with a key known to the server that was storing the database, but this would mean that someone who had access to that machine could access all the user keys. Another possibility is to encrypt the key in a way that can only be reconstructed with the cooperation of several independent machines. This is feasible, and we'll discuss it more in §24.9.1 *Key Escrow*.

Some applications don't require recoverable keys. An example of such an application is login. If a user loses the key required for login, the user can be issued a new key. A user may therefore want different keys for different uses, where only some of the keys are escrowed. For applications that do require recoverable keys, protection from compromise can be traded off against protection from loss.

1.12 VIRUSES, WORMS, TROJAN HORSES

> *Lions and tigers and bears, oh my*!
>
> —Dorothy (in the movie *The Wizard of Oz*)

People like to categorize different types of malicious software and assign them cute biological terms (if one is inclined to think of worms as cute). We don't think it's terribly important to distinguish between these things, but will define some of the terms that seem to be infecting the literature.

- **Trojan horse**—instructions hidden inside an otherwise useful program that do bad things. Usually the term Trojan horse is used when the malicious instructions are installed at the time the program is written (and the term *virus* is used if the instructions get added to the program later).
- **virus**—a set of instructions that, when executed, inserts copies of itself into other programs. More recently, the term has been applied to instructions in email messages that, when executed, cause the malicious code to be sent in email to other users.
- **worm**—a program that replicates itself by installing copies of itself on other machines across a network.

- **trapdoor**—an undocumented entry point intentionally written into a program, often for debugging purposes, which can be exploited as a security flaw.
- **logic bomb**—malicious instructions that trigger on some event in the future, such as a particular time occurring.
- **zombie**—malicious instructions installed on a system that can be remotely triggered to carry out some attack with less traceability because the attack comes from another victim. Often the attacker installs large numbers of zombies in order to be able to generate large bursts of network traffic.

We do not think it's useful to take these categories seriously. As with most categorizations (e.g., plants vs. animals, intelligence vs. instinct), there are things that don't fit neatly within these categories. So we'll refer to all kinds of malicious software generically as **digital pests**.

1.12.1 Where Do They Come From?

Where do these nasties come from? Except for trapdoors, which may be intentionally installed to facilitate troubleshooting, they are written by bad guys with nothing better to do with their lives than annoy people.

How could an implementer get away with writing a digital pest into a program? Wouldn't someone notice by looking at the program? One of the most famous results in computer science is that it is provably impossible to be able to tell what an arbitrary program will do by looking at it*, so certainly it would be impossible to tell, in general, whether the program had any unpleasant side effects besides its intended purpose. But that's not the real problem. The real problem is that nobody looks. Often when you buy a program you do not have access to the source code, and even if you did, you probably wouldn't bother reading it all, or reading it very carefully. Many programs that run have never been reviewed by anybody. A major advantage offered by the "open source" movement (where all software is made available in source code format) is that even if you don't review it carefully, there is a better chance that someone else will.

What does a virus look like? A virus can be installed in just about any program by doing the following:

- replace any instruction, say the instruction at location *x*, by a jump to some free place in memory, say location *y*; then
- write the virus program starting at location *y*; then

* This is known in the literature as *the halting problem*, which states that it is impossible in general to tell whether a given program will halt or not. In fact it is impossible in general to discern any nontrivial property of a program.

- place the instruction that was originally at location *x* at the end of the virus program, followed by a jump to x+1.

Besides doing whatever damage the virus program does, it might replicate itself by looking for any executable files in any directory and infecting them. Once an infected program is run, the virus is executed again, to do more damage and to replicate itself to more programs. Most viruses spread silently until some triggering event causes them to wake up and do their dastardly deeds. If they did their dastardly deeds all the time, they wouldn't spread as far.

How does a digital pest originally appear on your computer? All it takes is running a single infected program. A program posted on a bulletin board might certainly be infected. But even a program bought legitimately might conceivably have a digital pest. It might have been planted by a disgruntled employee or a saboteur who had broken into the computers of the company and installed the pest into the software before it was shipped. There have been cases where commercial programs were infected because some employee ran a program gotten from a friend or a bulletin board.

Often at holiday times people send email messages with attached programs and instructions to run them. While this used to require extracting the email message to a file and possibly processing it first, modern email systems make it very convenient to run such things... often just by clicking on an icon in the message. Often the result is some sort of cute holiday-specific thing, like displaying a picture of a turkey or playing a Christmas carol. It could certainly also contain a virus. Few people will scan the program before running it, especially if the message arrives from a friend. And if you were to run such a program and it did something cute, you might very well forward it to a friend, not realizing that in addition to doing the cute thing it might very well have installed a virus that will start destroying your directory a week after the virus is first run. A good example of a Christmas card email message is a program written by Ian Phillipps, which was a winner of the 1988 International Obfuscated C Code Contest. It is delightful as a Christmas card. It does nothing other than its intended purpose (I_1 have analyzed the thing carefully and I_2 have complete faith in me_1), but we doubt many people would take the time to understand this program before running it (see Figure 1-4).

Sometimes you don't realize you're running a program. PostScript is a complete programming language. It is possible to embed a Trojan horse into a PostScript file that infects files with viruses and does other damage. Someone could send you a file and tell you to display it. You wouldn't realize you were running a program by merely displaying the file. And if there was any hope of scanning a C program to find suspicious instructions, there are very few people who could scan a PostScript file and look for suspicious PostScript commands. PostScript is, for all practical purposes, a write-only language.

As mail systems get more clever in their attempt to make things more convenient for users, the risk becomes greater. If you receive a PostScript file and you are running a non-clever mail program, you will see delightful crud like Figure 1-5

```
/* Have yourself an obfuscated Christmas! */
#include <stdio.h>
main(t,_,a)
char *a;
{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);
}
```

Figure 1-4. Christmas Card?

```
%!PS-Adobe-3.0
%%Creator: Windows PSCRIPT
%%Title: c:\book\spcaut.doc
%%BoundingBox: 18 9 593 784
%%DocumentNeededResources: (atend)
%%DocumentSuppliedResources: (atend)
%%Pages: (atend)
%%BeginResource: procset Win35Dict 3 1
/Win35Dict 290 dict def Win35Dict begin/bd{bind def}bind def/in{72
mul}bd/ed{exch def}bd/ld{load def}bd/tr/translate ld/gs/gsave ld/gr
/grestore ld/M/moveto ld/L/lineto ld/rmt/rmoveto ld/rlt/rlineto ld
/rct/rcurveto ld/st/stroke ld/n/newpath ld/sm/setmatrix ld/cm/currentmatrix
ld/cp/closepath ld/ARC/arcn ld/TR{65536 div}bd/lj/setlinejoin ld/lc
/setlinecap ld/ml/setmiterlimit ld/sl/setlinewidth ld/scignore false
def/sc{scignore{pop pop pop}{0 index 2 index eq 2 index 4 index eq
and{pop pop 255 div setgray}{3{255 div 3 1 roll}repeat setrgbcolor}ifelse}ifelse}bd
```

Figure 1-5. Typical PostScript Code

If you wanted to display the file, you'd have to extract the mail message and send it to the printer, or input it into a special program that displays PostScript on your screen. However, a clever mail program might look at the message, recognize that it was PostScript, and automatically run the PostScript code to display the message. Although this is convenient, it is dangerous.

There are various other clever features being added to mail programs. Some mail programs allow the sender to send a program along with the mail message. Usually the mail message will arrive and display some sort of icon. If the receiver clicks on the icon, the transmitted program is executed. Someone, to illustrate this point, sent such a mail message. It displayed only as a button that said *push me*. When the person who received the message clicked on the box, it came up with a message that said, *I could have just reformatted your hard drive*.

In the first edition of this book, we said “Before the technology for clever mail goes much further, we ought to consider how we can reap the benefits of such cleverness while minimizing the security risks”. We can now confidently say that it has gone much further, no one has worried at all about security, and as anyone who has been stung by the email virus of the week can attest, the situation is a disaster.

There remain dangers associated with booting from floppy disks. If the hard drive of a computer were completely wiped clean, there has to be some way to come up again, so machines provide the feature that if there is a floppy in the drive when the machine is powered on, the machine boots off the floppy. This can be disabled, but it rarely is. Even if system software becomes sophisticated about security, it won’t be able to protect against Trojan horses on the boot device. When you turn on many machines with a floppy disk inserted into the drive (intentionally or accidentally), they execute the code on the floppy in a privileged mode. If there is a virus in that code, it can infect the system.

Most PCs are configured to detect a CD-ROM placed in the drive and will execute a startup program automatically when a new one is inserted. This may have been a relatively safe thing to do when writable CD-Rs were a rarity and most CD-ROMs were commercially manufactured, but today it is likely viruses will spread this way. With CD-RWs becoming common, viruses can spread across your CDs the way they once did across floppies.

1.12.2 Spreading Pests from Machine to Machine

How might a virus or worm spread from machine to machine? An infected program might be copied to a floppy or other medium and moved to another machine. Or, as we said, a mail message might carry the infection. But it is also possible for the pest to explore the network and send itself to other machines, once it is running in one machine on a network. Such a pest is called a worm.

One method a worm can employ to transmit itself to another machine is to actually log into the other machine. It can do this by guessing passwords. Sometimes things are set up to make it really easy; for instance, account name/password pairs might be stored in script files so that a naive user can access remote resources automatically. If the worm manages to find that information, it can easily log into other machines and install itself. Sometimes machines have trapdoor debugging features, such as the ability to run a command remotely. Sometimes it isn’t even necessary to log in to run such a command. And if intruders can execute commands, they can do anything.

> *The only reason all the computers in the world haven’t crashed at the same time is that they’re not all connected together yet.*
>
> —Dave Cheriton

In the first widely publicized example, a Christmas-card-type email message displayed a pretty animated tree on the screen, but also scanned the user's directory for all possible correspondents, including distribution lists, and mailed itself to them. Each time a recipient read a copy of the message, it sprang into life and mailed itself off to all the mailboxes it could locate from that node. This was written by someone who was not trying to attack systems—he was just trying to be "cute" but didn't consider the likely consequences of his action. Although this program did not destroy data, like a classic worm it completely overloaded and disabled the corporation's electronic mail.

Email borne viruses have become the most common kind in the last few years, taking advantage both of bugs in email systems and naive users. The "ILoveYou" virus was particularly virulent because users were willing to take a chance and run the attachment when the message had a seductive subject line.

Worms have also become more effective at spreading as more machines are continuously connected to the Internet via cable modems and DSL, but not carefully configured for security. In several cases, insecure default configurations or bugs in services that were enabled by default provided fertile ground for worms to spread.

1.12.3 Virus Checkers

How can a program check for viruses? There's rather a race between the brave and gallant people who analyze the viruses and write clever programs to detect and eliminate them, and the foul-smelling scum who devise new types of viruses that will escape detection by all the current virus checkers.

The most common form of virus checker knows the instruction sequence for lots of types of viruses, checks all the files on disk and instructions in memory for those patterns of commands, and raises a warning if it finds a match hidden somewhere inside some file. Once you own such a virus checker, you need to periodically get updates of the patterns file that includes the newest viruses. To evade detection of their viruses, virus creators have devised what is known as a **polymorphic virus** which changes the order of its instructions, or changes to functionally similar instructions, each time it copies itself. Such a virus may still be detectable, but it takes more work and typically requires a coding change and not just a new pattern file. Modern virus checkers don't just periodically scan the disk. They actually hook into the operating system and inspect files before they are written to disk.

Another type of virus checker takes a snapshot of disk storage by recording the information in the directories, such as file lengths. It might even take message digests of the files. It is designed to run, store the information, and then run again at a future time. It will warn you if there are suspicious changes. One virus, wary of changing the length of a file by adding itself to the program, compressed the program so that the infected program would wind up being the same length as the original. When the program was executed, the uncompressed portion containing the virus decompressed the rest of the program, so (other than the virus portion) the program could run normally.

It would be natural for viruses to attack the virus checkers rather than just trying to elude them. One could even imagine a virus spread by the mechanism a virus checker uses to keep its pattern files up to date. Such a virus has not been seen at the time of this writing, but it is something to look forward to.

1.12.4 What Can We Do Today?

The world was a scary place before computer viruses came along, and will most likely continue to be scary. How can you know whether you can trust a program not to do bad things when you run it? Wouldn't it be nice to have the equivalent of a lie-detector test for programs?

> *Ames slipped by a lie-detector test because no one had told the polygrapher he was under suspicion.*
>
> —Douglas Waller & Evan Thomas
> *Newsweek* article on CIA and traitor Aldrich Ames, October 10, 1994

Given that there is no infallible method to test a program for hidden bad side effects, we can't be completely safe, but there are some precautions that are worth taking:

- Don't run software from suspicious sources, like bulletin boards or people who aren't as careful as you are.
- Frequently run virus checkers. Have the industry employ people whose job it is to keep up with virus technology and come up with vaccines.
- Try to run programs in the most limited possible environments. For instance, if you have a PC in order to get real work done, and you also want to play games, sometimes using shareware or games copied from bulletin boards, have two machines. If you run a game with a virus, you'll only wipe out your games. A somewhat more practical way to accomplish this is to have a machine with multiple disks and a physical switch that connects only one of them at a time.
- When your system puts up a warning saying that something is dangerous, don't do it!
- Do frequent backups, and save old backups for a long time.
- Don't boot off floppies, except in an extreme circumstance, such as the first time you unpack your machine and turn it on. In those circumstances, be extremely careful about what floppy you boot from.

But mostly, the situation is pretty bleak, given the design of the operating systems for PCs. Maybe in the future, some of our suggestions in the next section might be implemented.

1.12.5 Wish List for the Future

I_2 always assumed computers were designed in such a way that no program that ran on the machine could possibly injure the machine. Likewise, it should not be possible to send a piece of information to a machine that might damage the machine. People are designed properly that way, aren't they?

> *Sticks and stones may break my bones but words will never hurt me.*
> (chant designed to encourage bullies to get physical)

But one of my_2 first programs consisted of something that just sat around and waited for interrupts, and processed the interrupts. The program itself, not counting the instructions for processing the interrupts, was `HERE: JUMP HERE` (that's not the exact instruction, because I_2 don't want to divulge the brand of computer). I_2 discovered (the hard way) that you weren't supposed to do that, because it burned out core at the instruction that kept getting executed over and over. Gerry Sussman, while a high school student, wrote a program that broke magnetic tapes. The guru who guarded and ran the mainframe didn't believe Gerry when he boasted that he could write a program to break tapes, so Gerry wrote his program to go down the entire row of tape drives, breaking the tape in each one. Another example was a machine designed with a small amount of nonvolatile memory, but the type that wore out after a finite number of writes, say 10000. That kind of memory is fine for something like saving terminal settings, since a human has to type the key sequence to cause a write, and a human won't do it very often. But if the same kind of memory is used for storing parameter settings received by a network management message, then it is possible to break the machine within seconds by sending it parameter settings over and over.

In an ideal world, it should be possible to load a floppy and examine the contents without fear. You should be able to receive any email message without fear. If it is a multimedia message, it should be possible to play the audio, display the video, print the text, or waft the odors without damage to either the machine or files stored on the machine. A file is just a bunch of bits. If the file claims to be audio, it should be possible, without risk of any type of harm, to play the file. Likewise, if the file claims to be something worthy of printing, it should not cause any harm to print the file.

Programs are a bit trickier. It should be possible to run a program and have confidence that it will not affect the files stored on your machine or the basic integrity of the operating system.

We know none of these "shoulds" are true today. The files on your machine can be virus-infected through email, by displaying a PostScript file, or simply by inserting a floppy disk or CD-ROM (on some machines). How could systems be designed more defensively?

One simple feature would be a write-protect switch on your hard disk. Sometimes you run programs that you know should not be writing to your hard disk. A game program shouldn't be writing to your hard disk. Perhaps it wants to record highest score, but consumers might be willing to do without that frill if it means that they can run any game they want without fear of wiping out their life's work. Legitimate game manufacturers could design their games to work with write-protected hard disks. There is still the risk that the social misfits who design games that spread viruses could design their games to work with write-protected hard disks, but have their program check to see if some user has forgotten to write-protect the disk, and then launch the virus.

Timesharing systems used to be much more defensive. You could not run a program that would write into some other user's memory, or modify any portion of the disk that you were not authorized to write into. But PCs make the assumption that there is only one human on the machine at a time, and that human ought to be able to do anything it wants. To make matters worse, to enable snazzy maximally flexible features, there are all sorts of surprising places where someone can insert a program that will be executed.

The operating system ought to be built more defensively. PCs should have accounts just like timesharing systems so that you can set up a game account that can't affect the rest of the system. Likewise, your normal account shouldn't be able to alter system software. You should be able to easily run a program with the right to access only a single directory (its own). The world is moving in this direction, but progress is slow.

1.13 THE MULTI-LEVEL MODEL OF SECURITY

Computer security has become sufficiently important that it was inevitable that governments would decide they needed to “do something about it”. And when governments want to know something about security, they turn to the experts: the military. And they develop standards and measurement tools by which security can be measured that are unbiased so as not to favor any one organization. And they mandate that anyone they can influence buy products meeting those standards.

The problem is that *secure* is not as simple to define as, say, *flame-retardant*. The security threats in different environments are very different, as are the best ways to counter them. The military has traditionally focussed on keeping their data secret (and learning the secrets of the other side). They are less concerned (though probably shouldn't be) about data getting corrupted or forged. In a paper world, forgeries are so difficult and so likely to expose the spies placing them that this threat takes a back seat. In the computerized environment, modification or corruption of data is a more likely threat.

1.13.1 Mandatory (Nondiscretionary) Access Controls

> *O negligence! … what cross devil made me put this main secret in the packet I sent the king? Is there no way to cure this? No new device to beat this from his brains?*
>
> —Shakespeare's *King Henry VIII*, act 3, scene 2

Discretionary means that someone who owns a resource can make a decision as to who is allowed to use (access) it. **Nondiscretionary access controls** enforce a policy where users might be allowed to use information themselves but might not be allowed to make a copy of it available to someone else. Strict rules are automatically enforced about who is allowed access to certain resources based on the attributes of the resource, and even the owners of the resources cannot change those attributes. The analogy in the paper world is that you might be given a book full of confidential information, but you are not allowed to take the book out of the building. In the military, information often has a security classification, and just because you have access to secret information does not mean you can forward it as you see fit. Only someone with the proper clearance can see it, and clearance levels are decided by a separate organization based on background investigations—not based on whether someone seems like a nice guy at lunch.

The basic philosophy behind discretionary controls is that the users and the programs they run are good guys, and it is up to the operating system to trust them and protect each user from outsiders and other users. The basic philosophy behind nondiscretionary controls is that users are careless and the programs they run can't be presumed to be carrying out their wishes. The system must be ever vigilant to prevent the users from accidentally or intentionally giving information to someone who shouldn't have it. Careless users might accidentally type the wrong file name when including a file in a mail message, or might leave a message world-readable. The concept is to confine information within a **security perimeter**, and thus not allow any information to move from a more secure environment to a less secure environment. A secure system would have both discretionary and nondiscretionary access controls, with the latter serving as a backup mechanism with less granularity.

Now of course, if you allow the users out of the building, there is an avenue for information to leak out of a secure environment, since a user can remember the information and tell someone once the user gets out of the security perimeter. There really is no way for a computer system to prevent that. But the designers wanted to ensure that no Trojan horse in software could transmit any information out of the perimeter, that nothing a user did inadvertently could leak information, and that users couldn't spirit out larger amounts of information than they could memorize.

1.13.2 Levels of Security

What does it mean for something to be "more sensitive" than something else? We will use a somewhat simplified description of the U.S. Department of Defense (DoD) definitions of levels of security as an example. It is a reasonably general model and similar to what is done in other contexts. It is sufficient to understand the security mechanisms we'll describe.

The security label of something consists of two components:

- A **security level** (also known as classification), which might be an integer in some range, but in the U.S. DoD consists of one of the four ratings unclassified, confidential, secret, and top secret, where unclassified < confidential < secret < top secret.
- A set of zero or more **categories** (also known as **compartments**), which describe kinds of information. For instance, the name CRYPTO might mean information about cryptographic algorithms, INTEL might mean information about military intelligence, COMSEC might mean information about communications security, or NUCLEAR might mean information about types of families.

Documents (or computer files) are marked with a security label saying how sensitive the information is, and people are issued security clearances according to how trustworthy they are perceived to be and what information they have demonstrated a "need to know."

A clearance might therefore be (SECRET;{COMSEC,CRYPTO}), which would indicate someone who was allowed to know information classified unclassified, confidential, or secret (but not top secret) dealing with cryptographic algorithms or communications security.

Given two security labels, (X,S_1) and (Y,S_2), (X,S_1) is defined as being "at least as sensitive as" (Y,S_2) iff $X \geq Y$ and $S_2 \subseteq S_1$. For example,

(TOP SECRET, {CRYPTO, COMSEC}) > (SECRET, {CRYPTO})

where ">" means "more sensitive than".

It is possible for two labels to be incomparable in the sense that neither is more sensitive than the other. For example, neither of the following are comparable to each other:

(TOP SECRET, {CRYPTO, COMSEC})

(SECRET, {NUCLEAR, CRYPTO})

1.13.3 Mandatory Access Control Rules

Every person, process, and piece of information has a security label. A person cannot run a process with a label higher than the person's label, but may run one with a lower label. Information is only allowed to be read by a process that has at least as high a rating as the information. The terminology used for having a process read something with a higher rating than the process is **read-up**. Read-up is illegal and must be prevented. A process cannot write a piece of information with a rating lower than the process's rating. The terminology used for a process writing something with a lower rating than the process is **write-down**. Write-down is illegal and must be prevented.

The rules are:

- A human can only run a process that has a security label below or equal to that of the human's label.
- A process can only read information marked with a security label below or equal to that of the process.
- A process can only write information marked with a security label above or equal to that of the process. Note that if a process writes information marked with a security label above that of the process, the process can't subsequently read that information.

The prevention of read-up and write-down is the central idea behind mandatory access controls. The concepts of confinement within a security perimeter and a generalized hierarchy of security classes were given a mathematical basis by Bell and La Padula in 1973 [BELL74]. There is significant complexity associated with the details of actually making them work. There has been significant subsequent research on more complex models that capture both the trustworthiness and the confidentiality of data and programs.

1.13.4 Covert Channels

A **covert channel** is a method for a Trojan horse to circumvent the automatic confinement of information within a security perimeter. Let's assume an operating system has enforced the rules in the previous section. Let's assume also that a bad guy has successfully tricked someone with a TOP SECRET clearance into running a program with a Trojan horse. The program has access to some sensitive data, and wants to pass the data to the bad guy. We're assuming the operating system prevents the process from doing this straightforwardly, but there are diabolical methods that theoretically could be employed to get information out.

The Trojan horse program cannot directly pass data, but all it needs is for there to be anything it can do that can be detected by something outside the security perimeter. As long as information can be passed one bit at a time, anything can be transmitted, given enough time.

One kind of covert channel is a **timing channel**. The Trojan horse program alternately loops and waits, in cycles of, say, one minute per bit. When the next bit is a 1, the program loops for one minute. When the next bit is a 0, the program waits for a minute. The bad guy's program running on the same computer but without access to the sensitive data constantly tests the loading of the system. If the system is sluggish, its conspirator inside the perimeter is looping, and therefore transmitting a 1. Otherwise, the conspirator is waiting, and therefore transmitting a 0.

This assumes those two processes are the only ones running on the machine. What happens if there are other processes running and stopping at seemingly random times (from the point of view of the program trying to read the covert channel)? That introduces noise into the channel. But communications people can deal with a noisy channel; it just lowers the potential bandwidth, depending on the signal to noise ratio.

Another kind of covert channel called a **storage channel** involves the use of shared resources other than processor cycles. For instance, suppose there were a queue of finite size, say the print queue. The Trojan horse program could fill the queue to transmit a 1, and delete some jobs to transmit a 0. The covert channel reader would attempt to print something and note whether the request was accepted. Other possible shared resources that might be exploited for passing information include physical memory, disk space, network ports, and I/O buffers.

Yet another example depends on how clever the operating system is about not divulging information in error messages. For instance, suppose the operating system says file does not exist when a file really does not exist, but says insufficient privilege for requested operation when the file does exist, but inside a security perimeter off limits to the process requesting to read the file. Then the Trojan horse can alternately create and delete a file of some name known to the other process. The conspirator process periodically attempts to read the file and uses the information about which error message it gets to determine the setting of the next bit of information.

There is no general way to prevent all covert channels. Instead, people imagine all the different ways they can think of, and specifically attempt to plug those holes. For instance, the timing channel can be eliminated by giving each security level a fixed percentage of the processor cycles. This is wasteful and impractical in general, because there can be an immense number of distinct classifications (in our model of (*one of four levels*, {*categories*}), the number of possible security perimeters is $4 \cdot 2^n$, where n is the number of categories).

Most covert channels have very low bandwidth. In many cases, instead of attempting to eliminate a covert channel, it is more practical to introduce enough noise into the system so that the bandwidth becomes too low to be useful to an enemy. It's also possible to look for jobs that appear to be attempting to exploit covert channels (a job that alternately submitted enough print jobs to fill the queue and then deleted them would be suspicious indeed if someone knew to watch). If the bandwidth is low *and* the secret data is large, and knowing only a small subset of the secret data is not of much use to an enemy, the threat is minimized.

How much secret data must be leaked before serious damage is done can vary considerably. For example, assume there is a file with 100 megabytes of secret data. The file has been transmitted, encrypted, on an insecure network. The enemy therefore has the ciphertext, but the cryptographic algorithm used makes it impossible for the enemy to decrypt the data without knowing the key. A Trojan horse with access to the file and a covert channel with a bandwidth of 1 bit every 10 seconds would require 250 years to leak the data (by which time it's hard to believe the divulging of the information could be damaging to anyone). However, if the Trojan horse had access to the 56-bit key, it could leak that information across the covert channel in less than 10 minutes. That information would allow the enemy to decrypt the 100-megabyte file. For this reason, many secure systems go to great pains to keep cryptographic keys out of the hands of the programs that use them.

1.13.5 The Orange Book

The National Computer Security Center (NCSC), an agency of the U.S. government, has published an official standard called “Trusted Computer System Evaluation Criteria”, universally known as “the Orange Book” (guess what color the cover is). The Orange Book defines a series of ratings a computer system can have based on its security features and the care that went into its design, documentation, and testing. This rating system is intended to give government agencies and commercial enterprises an objective assessment of a system’s security and to goad computer manufacturers into placing more emphasis on security.

The official categories are D, C1, C2, B1, B2, B3, and A1, which range from least secure to most secure. In reality, of course, there is no way to place all the possible properties in a linear scale. Different threats are more or less important in different environments. The authors of the Orange Book made an attempt to linearize these concerns given their priorities. But the results can be misleading. An otherwise A1 system that is missing some single feature might have a D rating. Systems not designed with the Orange Book in mind are likely to get low ratings even if they are in fact very secure.

The other problem with the Orange Book rating scheme is that the designers focused on the security priorities of military security people—keeping data secret. A rating of B1 or better requires implementation of multi-level security and mandatory access controls. In the commercial world, data integrity is at least as important as data confidentiality. Mandatory access controls, even if available, are not suitable for most commercial environments because they make some of the most common operations, such as having a highly privileged user send mail to an unprivileged user, very cumbersome.

Mandatory access controls do not by themselves protect the system from infection by viruses. Mandatory access controls allow write-up, so if some unprivileged account became infected by having someone carelessly run, say, a game program loaded from a bulletin board, the virus could spread to more secure areas. Ironically, if it was a very secure area that first got infected, the mandatory access control features would prevent the infection from spreading to the less secure environments.

The following is a summary of what properties a system must have to qualify for each rating.

D – Minimal Protection. This simply means the system did not qualify for any of the higher ratings; it might actually be very secure. No system is ever going to brag about the fact that it was awarded a D rating.

C1 – Discretionary Security Protection. The requirements at this level correspond roughly to what one might expect from a classic timesharing system. It requires

- The operating system must prevent unprivileged user programs from overwriting critical portions of its memory. (Note that many PC operating systems do not satisfy this condition.)
- Resources must be protected with access controls. Those access controls need not be sophisticated; classic owner/group/world controls would be sufficient.
- The system must authenticate users by a password or some similar mechanism, and the password database must be protected so that it cannot be accessed by unauthorized users.

There are additional requirements around testing and documentation, which become more detailed at each successive rating.

C2 – Controlled Access Protection. This level corresponds roughly to a timesharing system where security is an important concern but users are responsible for their own fates; an example might be a commercial timesharing system. The additional requirements (over those required for C1) for a C2 rating are

- access control at a per user granularity—It must be possible to permit access to any selected subset of the user community, probably via ACLs. An **ACL** is a data structure attached to a resource that specifies the resource's authorized users. C2 does not explicitly require ACLs, but they are the most convenient way to provide the granularity of protection that C2 requires.
- clearing of allocated memory—The operating system must ensure that freshly allocated disk space and memory does not contain "left-over" data deleted by some previous user. It can do that by writing to the space or by requiring processes to write to the space before they can read it.
- auditing—The operating system must be capable of recording security-relevant events, including authentication and object access. The audit log must be protected from tampering and must record date, time, user, object, and event. Auditing must be selective based on user and object.

It is reasonable to expect that C2-rateable systems will become ubiquitous, since they contain features that are commonly desired and do not represent an unacceptable overhead. It is somewhat surprising that such systems are not the norm.

B1 – Labeled Security Protection. Additional requirements at this level are essentially those required to implement Mandatory Access Controls for secrecy (not integrity) except that little attention is given to covert channels. Requirements for B1 above those for C2 include

- Security Labels: Sensitivity labels must be maintained for all users, processes, and files, and read-up and write-down must be prevented by the operating system.
- Attached devices must either themselves be labeled as accepting only a single level of information, or they must accept and know how to process security labels.
- Attached printers must have a mechanism for ensuring that there is a human-readable sensitivity label printed on the top and bottom of each page corresponding to the sensitivity label of the information being printed. The operating system must enforce this correspondence.

B2 – Structured Protection. Beyond B1, there are few new features introduced; rather, the operating system must be structured to greater levels of assurance that it behaves correctly (i.e., has no bugs). Additional requirements for B2 include

- trusted path to user—There must be some mechanism to allow a user at a terminal to reliably distinguish between talking to the legitimate operating system and talking to a Trojan horse password-capturing program.
- security level changes—A terminal user must be notified when any process started by that user changes its security level.
- security kernel—The operating system must be structured so that only a minimal portion of it is security-sensitive, i.e., that bugs in the bulk of the O/S cannot cause sensitive data to leak. This is typically done by running the bulk of the O/S in the processor's user mode and having a secure-kernel mini-O/S which enforces the mandatory access controls.
- Covert channels must be identified and their bandwidth estimated, but there is no requirement that they be eliminated.
- Strict procedures must be used in the maintenance of the security-sensitive portion of the operating system. For instance, anyone modifying any portion must document what they changed, when they changed it, and why, and some set of other people should compare the updated section with the previous version.

B3 – Security Domains. Additional requirements for B3 mostly involve greater assurance that the operating system will not have bugs that might allow something to circumvent mandatory access controls. Additional requirements include

- ACLs must be able to explicitly deny access to named individuals even if they are members of groups that are otherwise allowed access. It is only at this level that ACLs must be able to separately enforce modes of access (i.e., read vs. write) to a file.
- active audit—There must be mechanisms to detect selected audited events or thresholds of audited events and immediately trigger notification of a security administrator.
- secure crashing—The system must ensure that the crashing and restarting of the system introduces no security policy violations.

A1 – Verified Design. There are no additional features in an A1 system over a B3 system. Rather, there are formal procedures for the analysis of the design of the system and more rigorous controls on its implementation.

1.13.6 Successors to the Orange Book

> *Gee, I wish we had one of them Doomsday machines*
>
> —Turgidson, in *Dr. Strangelove*

Governments are rarely willing to adopt one another's ideas, especially if they didn't contribute. They would rather develop their own. The publication of the Orange Book in 1983 set off a series of efforts in various countries to come up with their own standards and classifications. These efforts eventually merged in 1990 into a single non-U.S. standard called ITSEC, followed by a reconciliation with the U.S. and the development of a single worldwide standard called the Common Criteria in 1994. Version 2.1 of the Common Criteria became an international standard in 1999.

The details of the various rating systems are all different, so passing the bureaucratic hurdles to qualify for a rating under one system would not be much of a head start toward getting an equivalent rating in another (much as different countries don't recognize each other's credentials for practicing medicine). The following table is an oversimplification that we hope won't be too offensive to the advocates of these rating systems, but it does allow the novice to judge what is being claimed about a system. In a partial acknowledgment of the multifaceted nature of security, many of the systems gave two ratings: one for the features provided and one for the degree of assurance that the system implements those features correctly. In practice, like the artistic and technical merit scores at the Olympics, the two scores tend to be closely correlated.

TCSEC (Orange Book)	German (Green Book)	British CLEF	ITSEC	Common Criteria
D	Q0		E0	EAL 0
				EAL 1
C1	Q1/F1	L1	C1/E1	EAL 2
C2	Q2/F2	L2	C2/E2	CAPP/EAL 3
B1	Q3/F3	L3	B1/E3	CSPP/EAL 4
B2	Q4/F4	L4	B2/E4	EAL 5
B3	Q5/F5	L5	B3/E5	EAL 6
A1	Q6/F5	L6	B3/E6	EAL 7

1.14 LEGAL ISSUES

The legal aspects of cryptography are fascinating, but the picture changes quickly, and we are certainly not experts in law. Although it pains us to say it, if you're going to do anything involving cryptography, talk to a lawyer.

1.14.1 Patents

One legal issue that affects the choice of security mechanisms is patents. Most cryptographic techniques are covered by patents and historically this has slowed their deployment. One of the important criteria for selection of the AES algorithm (see §3.5 *Advanced Encryption Standard (AES)*) was that it be royalty free.

The most popular public key algorithm is RSA (see §6.3 *RSA*). RSA was developed at MIT, and under the terms of MIT's funding at the time there are no license fees for U.S. government use. It was only patented in the U.S., and licensing was controlled by one company which claimed that the Hellman-Merkle patent also covered RSA, and that patent is international. Interpretation of patent rights varies by country, so the legal issues were complex. At any rate, the last patent on RSA ran out on September 20, 2000. There were many parties on that day.

To avoid large licensing fees, many standards attempted to use DSS (see §6.5 *Digital Signature Standard (DSS)*) instead of RSA. Although in most respects DSS is technically inferior to RSA, when first announced it was advertised that DSS would be freely licensable, i.e., it would not be necessary to reach agreement with the RSA licensing company. But the company claimed Hellman-Merkle covered all public key cryptography, and strengthened its position by acquiring rights to a patent by Schnorr that was closely related to DSS. Until the patents expired, the situation was murky.

> *"I don't know what you mean by* your way*," said the Queen: "all the ways about here belong to me…"*
>
> *—Through the Looking Glass*

Some of the relevant patents, luckily all expired, are:

- Diffie-Hellman: Patent #4,200,770, issued 1980. This covers the Diffie-Hellman key exchange described in §6.4 *Diffie-Hellman*.
- Hellman-Merkle: Patent #4,218,582, issued 1980. This is claimed to cover all public key systems. There is some controversy over whether this patent should be valid. The specific public key mechanisms described in the patent (*knapsack* systems) were subsequently broken.
- Rivest-Shamir-Adleman: Patent #4,405,829, issued 1983. This covers the RSA algorithm described in §6.3 *RSA*.
- Hellman-Pohlig: Patent #4,424,414, issued 1984. This is related to the Diffie-Hellman key exchange.

1.14.2 Export Controls

> *Mary had a little key*
> *(It's all she could export)*
> *And all the email that she sent*
> *Was opened at the Fort.*
>
> —Ron Rivest

The U.S. government used to impose severe restrictions on export of encryption. This caused much bitterness in the computer industry and led to some fascinating technical designs so that domestic products, which were legally allowed to use strong encryption, could use strong encryption where possible, and yet interoperate with exportable products that were not allowed to use strong encryption. We describe such designs in this book (see §19.13 *Exportability*). Luckily, export controls seem to be pretty much lifted. For historical reasons, and in case they become relevant again (we sure hope not), we couldn't bear to delete the following section. It was written for the first edition of this book, and was timely in 1995 when that edition was published:

The U.S. government considers encryption to be a dangerous technology, like germ warfare and nuclear weapons. If a U.S. corporation would like to sell to other countries (and the proceeds *are not going to be funding the Contras), it needs export approval. The export control laws around encryption are not clear, and their interpretation changes over time. The general principle is that the U.S. government does not want you to give out technology that would make it more difficult for them to spy. Sometimes companies get so discouraged that they leave encryption out of their products altogether. Sometimes they generate products that, when sold overseas, have the encryption mechanisms removed. It is usually possible to get export approval for encryption if the key lengths are short enough for the government to brute-force check all possible keys to decrypt a message. So sometimes companies just use short keys, or sometimes they have the capability of varying the key length, and they fix the key length to be shorter when a system is sold outside the U.S.*

Even if you aren't in the business of selling software abroad, you can run afoul of export controls. If you install encryption software on your laptop and take it along with you on an international trip, you may be breaking the law. If you distribute encryption software within the U.S. without adequate warnings, you are doing your customers a disservice. And the legality of posting encryption software on a public network is questionable.