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
```

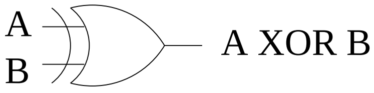
We append our remainder to the original string, giving $T(x) = 10110101101000$.

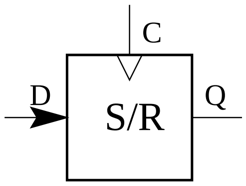
When this stream is received, it is divided but now will have no remainder if the stream is received without errors.

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.





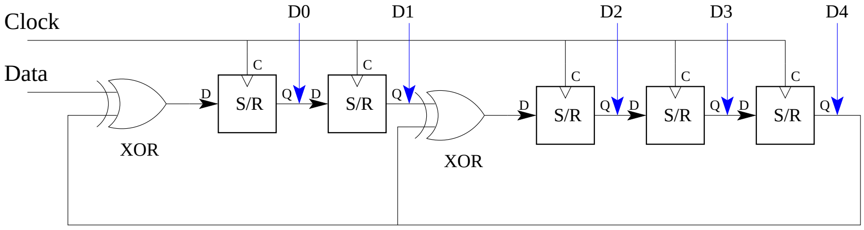
D	**C**	**Q**
0	↑	0
1	↑	1
0	↓	D
1	↓	D

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 single 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*** d single bit errors[6].
- We can ***correct*** up to d single 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 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:

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

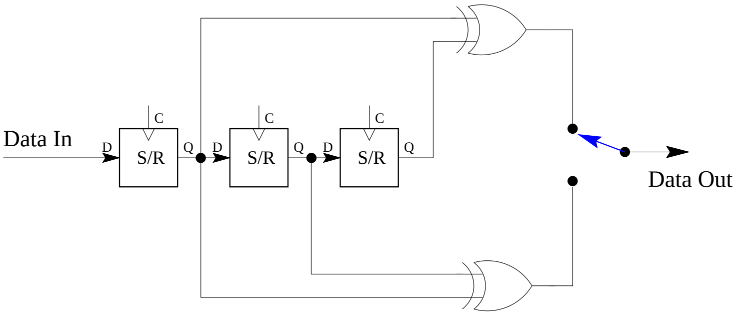
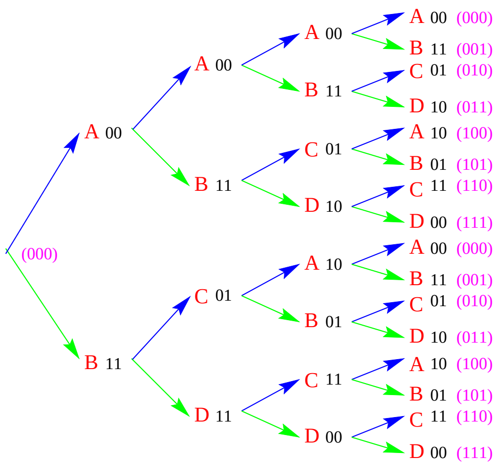


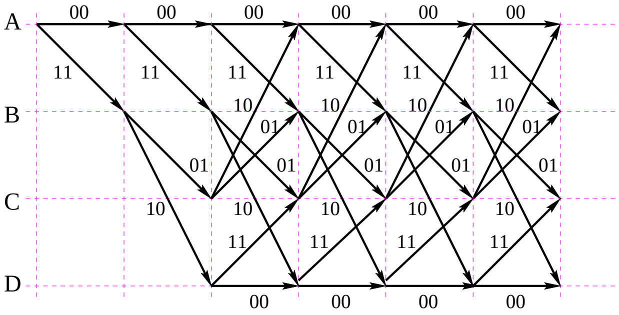
Figure 6.1: Sample convolutional encoder.

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.

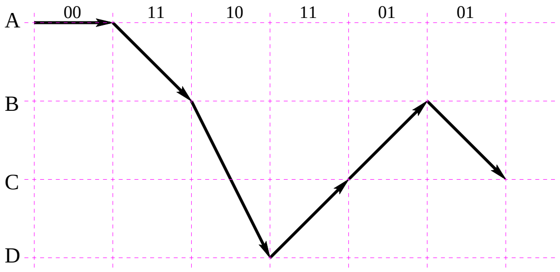
These four distinct states are labelled A, B, C and D in the diagram below[7].



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

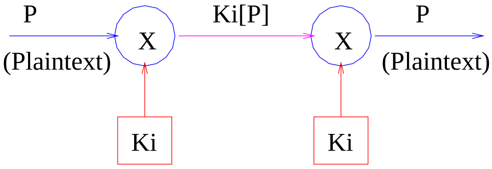


Figure 7.1: Symmetric key model

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.

```
Ifwecandiscoverthelengthoftherepeatedkeyandthetextislongenoughwecanjustconsiderthecipherttexttobeagroupofinterle
eaveIfwecandiscoverthelengthoftherepeatedkeyandthetextislongenoughwecanjustconsiderthecipherttexttobeagroupofint
---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

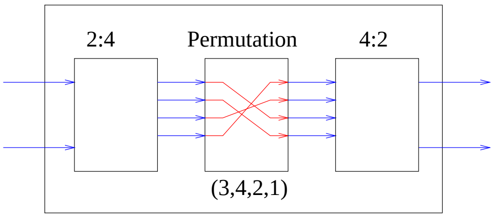


Figure 7.2: An S-box

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.

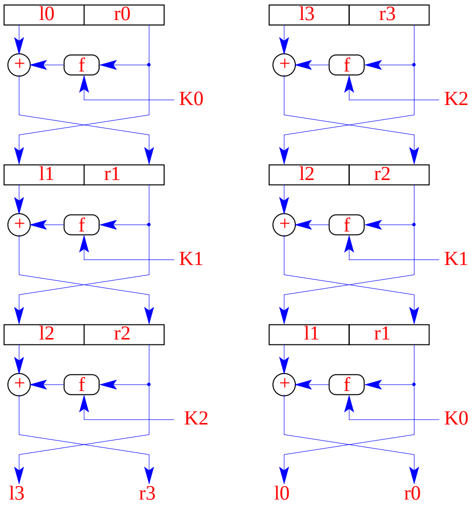


Figure 7.3: The Fiestel structure

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.

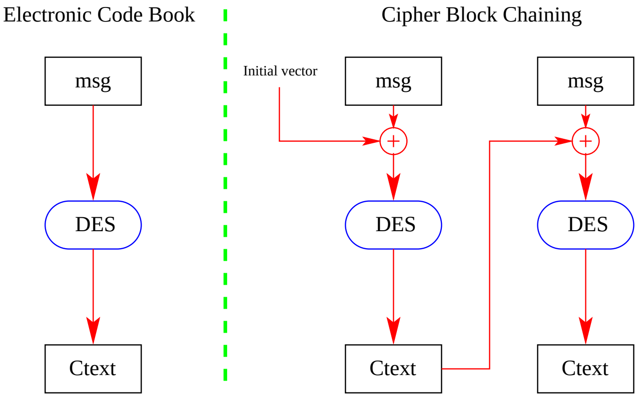


Figure 7.4: ECB and CBC

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)
```

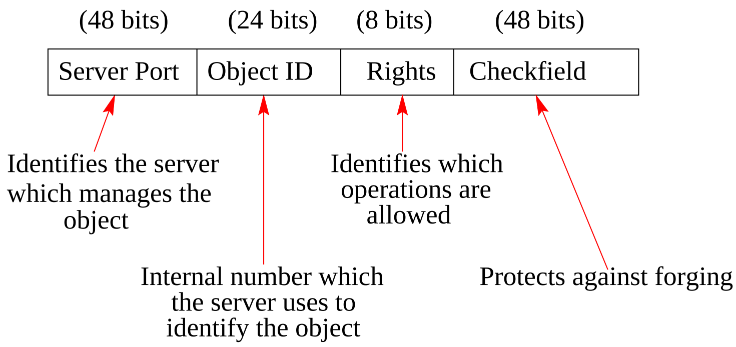
where the *encrypt* parameter determines if we are to encipher or decipher. The *schedule* contains the secret DES key.

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:



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.

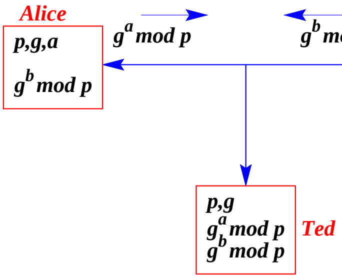


Figure 7.5: Diffie-Hellman key exchange protocol

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 ($g^a \;\mathbf{mod}\; p$ for Alice and $g^b \;\mathbf{mod}\; p$ for Bob).
- Ted knows g, p, $g^a \;\mathbf{mod}\; p$ and $g^b \;\mathbf{mod}\; p$, 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	722,600,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000

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

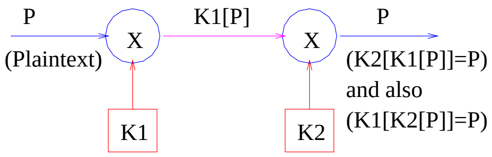


Figure 7.6: Encryption using public keys

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

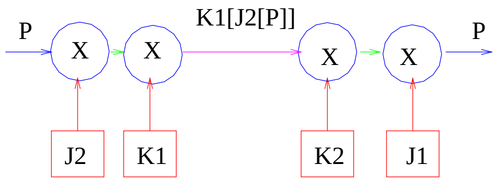


Figure 7.7: Authentication using public keys

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)

```
2519590847565789349402718324004839857142928212620403202777713783604366202070759555626401852588078440691829064124951508
2189298559149176184502808489120072844992687392807287776735971418347270261896375014971824691165077613379859095700097330
4597488084284017974291006424586918171951187461215151726546322822168699875491824224336372590851418654620435767984233871
8477444792073993423658482382428119816381501067481045166037730605620161967625613384414360383390441495263443219011465754
4454178424020924616515723350778707749817125772467962926386356373289912154831438167899885040445364023527381951378636564
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 P by:

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

To decode C back to P:

1. Calculate $P = C^D \,\mathbf{mod}\, N$.

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.