1

Ciphers and Fundamentals

1.1 Introduction

The future of the Internet, especially in expanding the range of applications, involves a much deeper understanding of privacy, integrity checking and authentication. Without this the Internet cannot properly expand and be trusted in its provision of services. One of the best ways to preserve privacy, check integrity and prove identity is data encryption, and which is known as the science of cryptographics.

Within encryption we often define the concept of Bob and Alice, who are involved with the communications, and Eve, who could listen or even modify their communications, or who could even pretend to be them. Bob and Alice thus communicate over a communication channel and which Eve is likely to have access to. In a secure environment Bob and Alice should be able to communicate freely, and identify themselves to each other, without Eve ever being able to reveal any of the messages involved, or being able to pretend to be them (Figure 1.1). The process typically involves taking some **plaintext**, and then converting it into **ciphertext**, which Eve should not be able to interpret, and then to convert it back into plaintext. Normally the conversion of plaintext to ciphertext is known as encryption, and the reverse is known as decryption.

In order to keep things secret, the two main methods that Bob and Alice can use are:

- A unique algorithm. This is an algorithm that both Bob and Alice know, but do not tell Eve. The algorithm for encoding and decoding is thus kept secret.
- Use a well-known algorithm. In this method Eve knows the algorithm, but where Bob and Alice use a special electronic key to uniquely define how the message is converted into cipertext, and then back again.

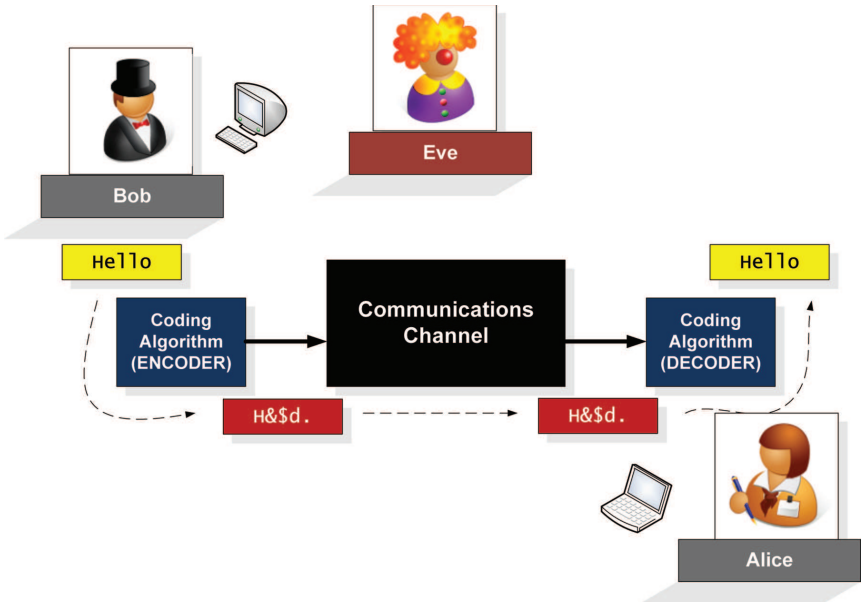


Figure 1.1 Bob, Alice and Eve.

A particular problem in any type of encryption is the passing of the information to define the secret, such as for the algorithm to be used or for an electronic key, as Eve may be listening to their communications.

This chapter looks at some of the basic principles of encryption, with the following chapters investigating the usage of secret-key (symmetric encryption) and public-key (asymmetric) methods. In secret-key encryption, we use a single electronic key to encrypt the plaintext, and the same key is then used to decrypt (normally involving a reversing of the encryption process). For public-key methods, we generate two electronic keys, and of which one is used encrypt the plaintext, and the other is used to decrypt it back to plaintext.

The concept of secret key encryption can be likened to Bob and Alice using a lockable box, of which only they have the key. Unfortunately, neither Bob nor Alice knows if Eve has taken a copy of their key. With public key encryption, Bob can create a number of identical padlocks, of which only he has the key to open them. Then if Alice wants to send him something, she will use one of his padlocks, and lock the box. Eve will thus not be able to open the box, as she will not have the required key. Bob must, obviously, keep the key to the padlock safe, so that Eve can't get access to it. The padlock can then be defined as his public key, and the other key as his private key.

As we will find, public and secret key methods often work together in perfect harmony, where secret key methods provide the actual core encryption, and public key methods provides ways to authenticate identities, and to pass encryption keys.

1.2 Simple Cipher Methods

One method of converting a message into cipher text is for Bob and Alice to agree on an algorithm which Bob will use to scramble his message, and then for Alice to do the opposite in order to unscramble the scrambled message. Thus, as long as Eve does not know the scrambling method, the cipher text will be secure. For example if Bob and Alice are sitting in a room where Eve is present, and then Bob taps on the table with a series of short (di) or long taps (dit), Bob can then pass a secret message to Alice, as long as they have agreed on what the codes identify. He might thus tap *di-di-di-dit*, and then pauses and taps *di-dit*, and where he passes the message in a standard Morse Code alphabet. In this way Alice decodes the message as "hi". Eve might eventually see that Bob is passing a message to Alice, but needs to know the type of encoding that they are using. In this way Bob and Alice agree on their method before they encode their messages, but where Eve may have heard them discussing the method that will be used. Eve, could also analyse their transmissions and then determine the codes by looking at common patterns.

With cipher methods we can use a mono-alphabetic code, where we create a single mapping from our alphabet to a cipher alphabet. This type of alphabet coding remains constant, whereas a polyalphabet can change its mapping depending on a variable keyword.

1.2.1 Morse Code

In a time when it was only possible to send sound pulses through a communications channel, Samuel F. B. Morse created a code mapping which sent pulses of electric current along wires with a silence in-between. Morse code is thus an encoding method, rather than a cipher, and works by translating characters into sequences of dots (.) and dashes (-). When transmitted as a sound pattern the dash lasts around three times longer than a dot, and with a longer delay between words as there is between letters.

The code was designed so that each of the characters varies in length approximately with the occurance of the letter in common English (Figure 1.2). For example there is a short code for an 'e' (dot), and a longer

one for a less common letter, such as 'j' (dot dash dash dash). For many years Morse code was used by radio operators, and provided the standard sequence of a ship in distress: Dot Dot Dot ...Dash Dash Dash ...Dot Dot Dot (or SOS).

📖 **Web link (Morse code):** http://asecuritysite.com/coding/morse

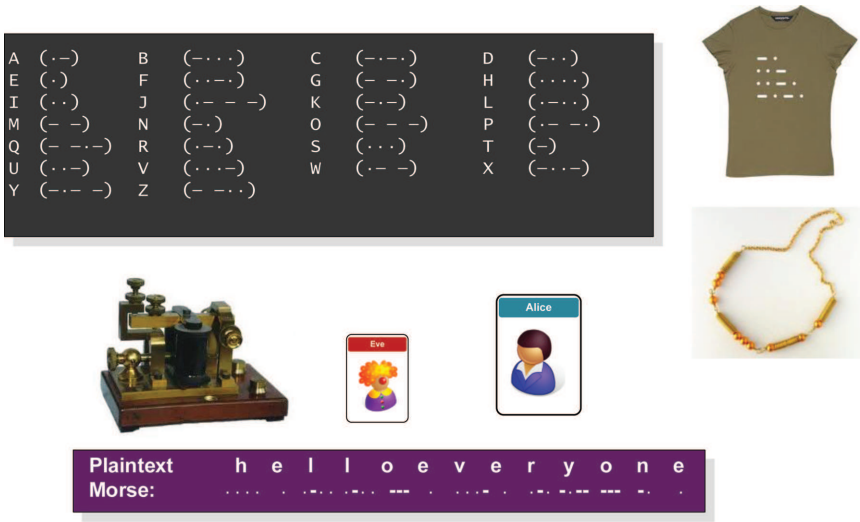


Figure 1.2 Morse code.

📖 **Web link (Morse code):** http://asecuritysite.com/encryption/morse

As an extension, the Fractionated Morse Cipher uses a 26 character key mapping and converts a plaintext input to Morse code. It then converts this into fixed-length chunks of Morse code, which are then converted into ciphertext letters. In converting the plaintext to Morse code, it uses 'x's between characters and 'xx' between words. For example, "Hello World" is Morse Code is:

```
.... . .–.. .–.. --- /       .–– --- .–. .–.. –..
H    E   L    L   O  SPACE    W  O   R   L    D
```

We can then make this into a string with an 'x' between characters:

```
Plain text:   H    e l    l    o    w    o   r    l    d
Morse string: ....x.x.–..x.–..x---xx.––x---x.–.x.–..x–..
```

We can now use three-character mappings to convert these back to text:

```
['...','.x.','x.-','..x',.-.','.x-','---x','x.-','-x-',
'---x','.-.','x.-','..x,'-..']
```

and where the mapping are:

```
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
 . . . . . . . . . . - - - - - - - - - - x x x x x x x
 . . . - - - x x x . . . - - - x x x . . . - - - x x
 . - x . - x . - x . - x . - x . - x . - x . - x . -
```

We can then use this mapping (such as A is defined as '...', B as '..-' and C as '..x'. Next we can convert them back with:

```
AGTCDHOTQODTCJ
```

For "Peter piper picked" we get:

```
.--.x.x-x.x.-.xx.--.x..x.--.x.x.-.xx.--.x..x-.-.x-.-x.x-..xx
P    e t e r ' ' p    i   p  e   r' ' p    i   c    k   e   d '
```

📖 **Web link (Fractionated Morse code):** http://asecuritysite.com/encryption/frac

1.2.2 Pigpen

Within ciphers, it is useful if Bob and Alice can create a cipher mapping that is easy to remember. One of the best methods is to use a graphical method, as the human eye often finds it easier to map graphical characters than to map alphabetic ones. The Pigpen cipher is a good example of this and uses a mono-alphabet substitution method.

For the Pigpen cipher, we initially created four grids in a square and a diagonal shape, with a dot placed in the second grid version (Figure 1.3). Next the alphabet characters are laid-out in sequence within the grids. Figure 1.4 outlines the mapping of the plaintext string of "biometric".

The problem with Pigpen is that once the mapping is known, it is difficult to keep the message secret. Bob could, though, embed it into a valid looking graphic, and send it to Alice. Eve, then, might not be able to see the embedded Pigpen symbols, but where Alice knows where to look.

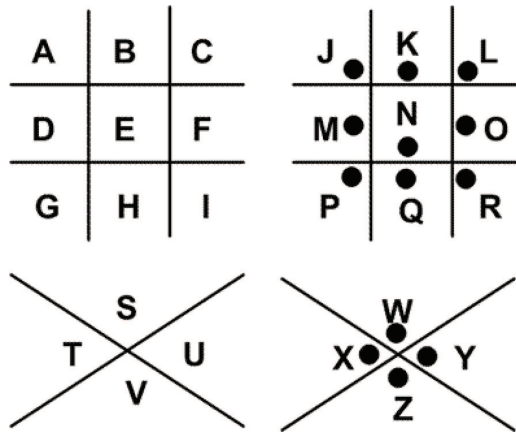


Figure 1.3 Code mapping.

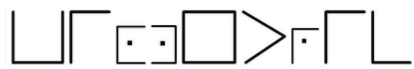


Figure 1.4 Code mapping.

🕮 **Web link (Pigpen code):** http://asecuritysite.com/challenges/pigpen

1.2.3 Rail Code

A useful method of hiding the cipher method is to scramble the plaintext letters in some way, and where it is not possible for the human eye to spot a pattern. Someone who knows the method will then be able to quickly decode. One method which scrambles in a defined pattern is the rail fence cipher. With this the message is written in a sequence across a number of rails. For example, if we use three rails, with a message of 'WE ARE DISCOVERED. FLEE AT ONCE', we get:

```
W . . . E . . . C . . . R . . . L . . . T . . . E
. E . R . D . S . O . E . E . F . E . A . O . C .
. . A . . . I . . . V . . . D . . . E . . . N . .
```

and where we then read across the rails to give a cipher code of "WECRL TEERD SOEEF EAOCA IVDEN". When we reverse of the process, we count the number of characters in the cipher, and map out with an 'X' for a position on the rail. The cipher is then written in sequence across the rails. So, for

example, if we have cipher text of "AALHP", we write out for five missing characters:

```
X . . . X
. X . X .
. . X . .
```

and next we layout across each row:

```
A . . . A
. L . H .
. . P . .
```

which we can then read as "alpha".

📖 **Web link (Rail fence):** http://asecuritysite.com/challenges/rail

1.2.4 BIFID Cipher

The BIFID cipher uses a grid and was invented by Felix Delastelle in 1901. In its simplest form it creates a grid and which maps the letters into numeric values. In creating the grid, we scramble the alphabetic characters, such as:

```
  1 2 3 4 5
1 B G W K Z
2 Q P N D S
3 I O A X E
4 F C L U M
5 T H Y V R
```

Next we look up the grid, and then arrange the two-character value into two rows. For example is we have a plaintext of "maryland", then "m" is "4" and "5", so we place "4" in the first row, and "5" in the second row, and continue to do this for all the letters:

```
maryland
43554322
53533334
```

Next we read along the rows and merge, to give:

```
43 55 43 22 53 53 33 34
```

And finally we convert them back to letters from the grid:

```
L R L P Y Y A X
```

Let's try the reverse, with DXETE, and when looking at the grid we get:

```
24 34 35 51 35
```

We can then put then into rows to give:

```
2 4 3 4 3
5 5 1 3 5
```

This gives us 25 (s) 45 (m), 31 (i), 43 (l) and 35 (e) – which is "smile".

📖 **Web link (Bifid cipher):** http://asecuritysite.com/coding/Bifid

We can make the grids more complex, such as with the four-square cipher. This method uses four 5×5 matrices arranged in a square, are where each matrix contains 25 letters. The upper-left and lower-right matrices are the "plaintext squares" and each contains a standard alphabet. The upper-right and lower-left squares are the "ciphertext squares" and have a mixture of characters.

First we break the message into bigrams, such as with "ATTACK AT DAWN" which gives:

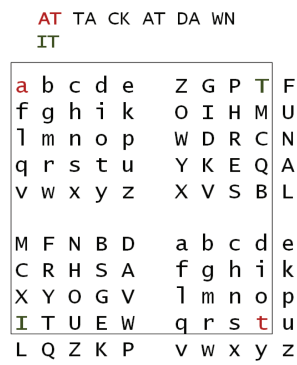
```
AT TA CK AT DA WN
```

We now use the four squares and locate the bigram to cipher in the plain alphabet squares. With 'AT', we take the first letter from the top left square, the second letter from the bottom right square:

```
a b c d e   Z G P T F
f g h i k   O I H M U
l m n o p   W D R C N
q r s t u   Y K E Q A
v w x y z   X V S B L
```

```
M F N B D   a b c d e
C R H S A   f g h i k
X Y O G V   l m n o p
I T U E W   q r s t u
L Q Z K P   v w x y z
```

Now, we determine the characters in the ciphertext around the corners of the rectangle for 'AT':



And so we pick off 'TI'. The result becomes:

```
ATTACKATDAWN
TIYBFHTIZBSY
```

📖 **Web link (Four square cipher):** http://asecuritysite.com/challenges/four

1.2.5 Playfair

The Playfair cipher was created by Charles Wheatstone, but was made famous by Lord Playfair. Initally a grid is created with a secret phrase, such as:

```
napierrun
```

Next we write out the 5 × 5 matrix, but do not repeat characters (and get rid of 'J'):

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

If we use the phrase of "GREATS", we split into sequences of two character sets:

```
GR EA TS
```

The rules are then:

[1] If they are in different columns, take from the rectangle defined between them and pick off the opposite ends.
[2] If they are in the same column, select the letter one below (and wrap-round if necessary).
[3] If they are in the same row, select the letter one along (and wrap-round if necessary).

The cipher is then created with:

- 'G','R' are bounded by 'FU' (Rule 1).
- 'E', 'A' are in the same row so we select one letter along (Rule 3) to give 'NP'.
- 'T','S' are in the same row so we select one letter along (Rule 3) to give 'MT'.

The cipher is thus "FUNPMY" (Figure 1.5).

📖 **Web link (Playfair):** http://asecuritysite.com/coding/playfair

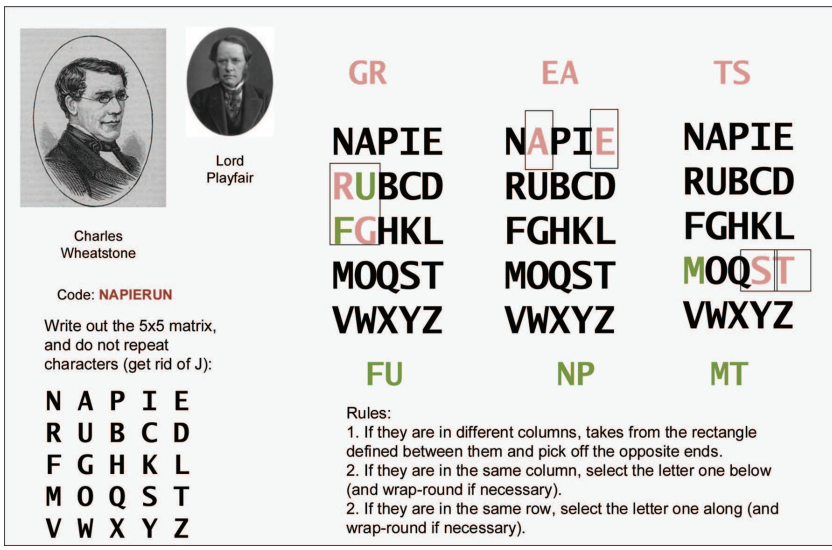


Figure 1.5 Code mapping.

1.2.6 Homophonic Substitution Code

Cipher codes can often be analysed using the probability of the letters/symbols in the ciphertext. A homophonic substitution code aims to overcomes this problem, as it varies the number of codes assigned to each character, and relates this to the probability of the characters. For example the character 'e' might have 12 codes assigned to it, but 'z' would only have one. An example code is given in Table 1.1. The homophonic substitution code is monoalphabet, even though it as uses one translation for the code mappings, as several codes can be used for a single plaintext letter.

With this, each of the codes is randomly assigned, with the number of codes assigned relating to the probability of their occurrence. Thus, using the code table in Table 1.1, the code mapping for "helloeveryone" would be:

```
Plaintext    h  e  l  l  o  e  v  e  r  y  o  n  e
Ciphertext: 19 25 42 81 16 26 22 28 04 55 30 00 32
```

In this case there are four occurrences of the letter 'e', and each one has a different code. As the number of codes depends on the number of occurrences of the letter, where each code will roughly have the same probability. It will thus be difficult to determine the code mapping from the probabilities of codes. Unfortunately the code is not perfect as the English language still contains certain relationships which can be traced. For example the letter 'q' would be represented by a single code, but there is a high probability that the next character will be a 'u'. Thus, using Table 1.1, there would only be three codes which would follow the value for a 'q'. If a given ciphertext contains a code followed by one of three codes, then it is likely that the plaintext is a 'q' and a 'u'.

📖 **Web link (Homophonic cipher):** http://asecuritysite.com/coding/ho

Table 1.1 Example homophonic substitution

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
07	11	17	10	25	08	44	19	02	18	41	42	40	00	16	01	15	04	06	05	13	22	45	12	55	47
31	64	33	27	26	09	83	20	03			81	52	43	30	62		24	34	23	14		46		93	
50		49	51	28			21	29			86		80	61			39	56	35	36					
63			76	32			54	53			95		88	65			58	57	37						
66				48			70	68					89	91			71	59	38						
77				67			87	73						94			00	90	60						
84				69										96					74						
				72															78						
				75															92						
				79																					
				82																					
				85																					

1.2.7 Caesar Coding and Scrambled Alphabet

The problem we have with encoding methods is that they can often be cracked with a simple lookup between the plaintext value and the equivalent cipher code. To make it more difficult we need to create a cipher which has a shared secret, and which only Bob and Alice know. The cipher text then changes with respect to this secret. An example of this is with the Caesar cipher, and which was created by Julius Caesar (using a 3-letter shift). Bob and Alice then simply agree to the number of shifts that the alphabet needs to be moved by.

In the example in Figure 1.6 the letters for the cipher have been moved forward by two positions, where a 'c' becomes an 'A'. Thus 'the' will be coded as 'RFC'. There are, though, several problems with this type of coding. The main one is that it is not secure as there are only 25 unique codings, and will thus be fairly easy for someone to find the mapping.

An improvement is to scramble the mapping using a code mapping (Figure 1.7), and where a random mapping is used to determine the cipher mappings. For the first character to be mapped ('a'), we would have 26 possible mapping. If we then move to the next character mapping ('b') we would have 25 remaining possible mappings. We can then continue on and would end up with:

26! mappings which gives approximately 4.03×10^{26} mappings

As we now have many more possible mappings, the cipher becomes more secure, as it is likely that Eve will have to search through many mappings until she finds the right one. This type of approach is know as a **brute force** method, as Eve tries all the possible code mappings, until she finds a solution. The worst attempt will see her search through all the possible mappings, but she might also find it on her first attempt. On average, though, she will search through half the possible mappings to find the right solution.

To work out how long she will take, we can assume that, on average, she will search through half of the code mappings. So if she takes one second to check each mapping, the time taken, on average, will be:

$T_{average} = (4.03 \times 10^{26})/2$ seconds

which is around 6.4×10^{18} years (over 6,400,000,000,000,000,000 years).

The scrambled alphabet code thus looks secure from a brute force viewpoint. Unfortunately it can be cracked fairly quickly by using frequency analysis. For the code in Figure 1.7, we can see 'A' in the cipher appears most often, and since 'e' is the most popular English letter, it is likely that it maps to a plaintext 'e'. Next we can see that 'Q' appears four times, thus it is most likely to be mapped to a 't', which is the next most probable letter in the English alphabet.

A more formal analysis of the probabilities is given in Table 1.2 and where we can see that the letter 'e' is the most probable, followed by 't', and then 'o', and so on. Along with analysing single letter occurrences, it is also possible to look at two-letter occurrences (*digrams*), or even three-letter occurrences (*trigrams*). We could also analyse the occurrences of words (which are separated by spaces), and where 'the' is the most common word.

A scrambled alphabet cipher scheme is easy to implement, but, unfortunately, once it has been 'cracked', it is easy to decrypt the ciphered data. Normally, to improve the cipher process, the cipher has extra parameters which change the mapping. This might include changing the mapping over time, such as for the time-of-day or the date. In this way, Bob and Alice would know the mappings of the code for a given time and/or date, such as having different mapping for each day of the week. The Enigma machine was used in the War where operators re-configured the machines every day with a code book (or key sheet). Each key sheet contained the daily Enigma settings over the period of a month, and where the machine was reconfigured each day.

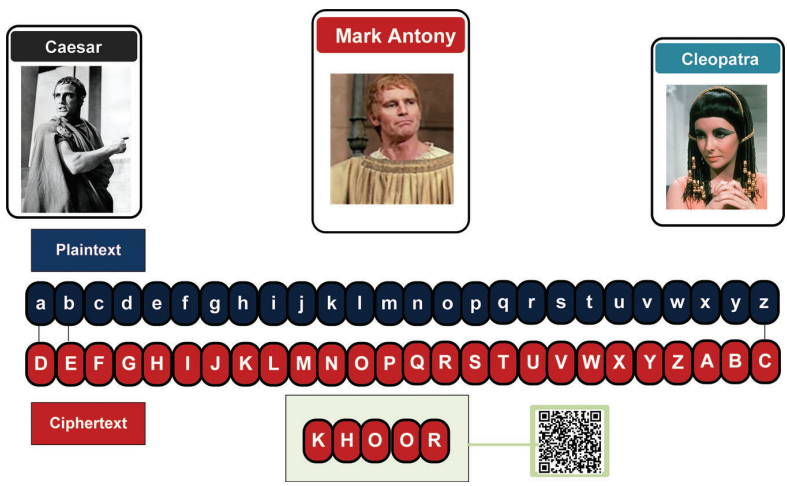


Figure 1.6 Caesar code.

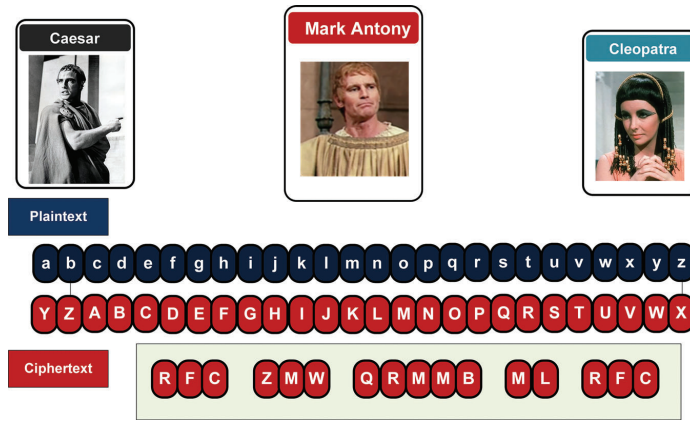


Figure 1.7 Code mapping.

Table 1.2 Probability of occurrences

Letters (%)		Digrams (%)		Trigrams (%)		Words (%)	
E	13.05	TH	3.16	THE	4.72	THE	6.42
T	9.02	IN	1.54	ING	1.42	OF	4.02
O	8.21	ER	1.33	AND	1.13	AND	3.15
A	7.81	RE	1.30	ION	1.00	TO	2.36
N	7.28	AN	1.08	ENT	0.98	A	2.09
I	6.77	HE	1.08	FOR	0.76	IN	1.77
R	6.64	AR	1.02	TIO	0.75	THAT	1.25
S	6.46	EN	1.02	ERE	0.69	IS	1.03
H	5.85	TI	1.02	HER	0.68	I	0.94
D	4.11	TE	0.98	ATE	0.66	IT	0.93
L	3.60	AT	0.88	VER	0.63	FOR	0.77
C	2.93	ON	0.84	TER	0.62	AS	0.76
F	2.88	HA	0.84	THA	0.62	WITH	0.76
U	2.77	OU	0.72	ATI	0.59	WAS	0.72
M	2.62	IT	0.71	HAT	0.55	HIS	0.71
P	2.15	ES	0.69	ERS	0.54	HE	0.71
Y	1.51	ST	0.68	HIS	0.52	BE	0.63
W	1.49	OR	0.68	RES	0.50	NOT	0.61
G	1.39	NT	0.67	ILL	0.47	BY	0.57
B	1.28	HI	0.66	ARE	0.46	BUT	0.56
V	1.00	EA	0.64	CON	0.45	HAVE	0.55
K	0.42	VE	0.64	NCE	0.43	YOU	0.55
X	0.30	CO	0.59	ALL	0.44	WHICH	0.53
J	0.23	DE	0.55	EVE	0.44	ARE	0.50
Q	0.14	RA	0.55	ITH	0.44	ON	0.47
Z	0.09	RO	0.55	TED	0.44	OR	0.45

Refer to the following pages:

🕮 **Web link (Caeser code):** http://asecuritysite.com/coding/caeser
🕮 **Web link (Scrambled code):** http://asecuritysite.com/coding/scramble
🕮 **Web link (Scrambled code challenge):** http://asecuritysite.com/challenges/scramb

1.2.8 Vigenère Cipher

An improved code over the scrambled alphabet approach was developed by Vigenère, where a different mapping, based on a keyword, is used for each character of the cipher. This is known as a *polyalphabetic* cipher as it uses a number of cipher alphabets. The way that the cipher mapping changes is agreed by Bob and Alice. One of the most popular methods is to use a code word which they agree on, and then move the mapping based on the characters in the keyword.

For example, if we use the mapping of Table 1.3, and if the code word is "GREEN", then the rows used are: Row 6 (G), Row 17 (R), Row 4 (E), Row 4 (E), Row 13 (N), Row 6 (G) and Row 17 (R). The message of "hellohowareyou" is thus converted as:

Keyword	`GREENGREENGREE`
Plaintext	`hellohowareyou`
Ciphertext	`NVPPBNFAEEKPSY`

The great advantage of this type of cipher is that the same plaintext character is likely to be coded to different mappings, depending on the position of the keyword. For example, for a keyword of GREEN, 'e' can be coded as 'K' (for G), 'V' (for R), 'I' (for E) and 'R' (for N). The method, though, was cracked by Major Friedrich Wilhelm Kasiski, a German infantry officer. He was the first to propose a method of attacking polyalphabetic substitution ciphers, and, in 1863, published a 95-page book on cryptography:

> Die Geheimschriften und die Dechiffrir-Kunst "Secret writing and the Art of Deciphering"

Its main focus was on the Vigenère cipher and where he developed a method known as Kasiski examination. In it he analysed the gaps between repeated ciphertext fragments, so that he could gain a hint on the key length. In this, we take the cipher message and analyse for repeated patterns, and which

gives a hint towards the key size. For example, if we have a message of "theywillnotkeeptheburningdeck" and then with a key of "abc", we get:

```
theywillnotkeeptheburningdeck
abcabcabcabcabcabcabcabcabcab
TIGYXKLMPOUMEFRTIGBVTNJPGEGCL
```

We can see that the "the" word has aligned to the key:

```
 the  ywillnotkeep  the   burningdeck
 abc  abcabcabcabc  abc   abcabcabcab
 TIG  YXKLMPOUMEFR  TIG   BVTNJPGEGCL
```

So we could reason that we might have a key size of three. Normally, though, we need a considerable amount of cipher text to accurately guess the key size. We can then use a frequency analysis method to get a shortlist for the possible key values.

📖 **Web link (Kasiski analysis):** http://asecuritysite.com/encryption/kasiski

📖 **Web link (Vigenère analysis):** http://asecuritysite.com/encryption/vig_crack

To improve security, the greater the size of the code word, the more the rows that can be included in the cipher process. It is also safe from analysis of common two- and three-letter occurrences, if the keysize is relatively long. For example 'ee' could be encrypted with 'KV' (for GR), 'VI' (for RE), 'II' (for EE), 'IR' (for EN) and 'RK' (for NG).

1.2.9 One-Time Pad (OTP)

The problem with the ciphers previously defined is that once Eve knows the method, she can normally crack all the codes created. Also if Bob and Alice use a keyword, Eve could try lots of different keywords to see if one works. In this way most ciphers can be broken, and where it is just a matter of time before it is cracked. If the time relevance of the message is greater than the average time to crack, the provenance of message can be preserved. For example if an army sends a message of "ATTACK" to their troops, the message just has been be secret until the time that they attack. After they have attacked then it would not matter if their cipher was cracked.

If we want an uncrackable cipher, we must use a one-time pad, and which is a cipher code mapping that is used only once. The one-time mapping is

Table 1.3 Coding

Row	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
19	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

then shared between Bob and Alice, and is used once to send a message, and where another pad is created for another message (Figure 1.8).

For example, we can first create a code book, which only Bob and Alice know:

```
yehq  medlg yaqif xygfs vlznx
llyyk ikbsy tvoon nvtuq qzvvn
ucyio nftsj bffbx ozxkl ckrsf
asfxg mqdlp gltek obvfm hqrxc
rbljl jlgcn vzwlw kctlq cftzx
bpmgy kaiup lftaf ufqrp ofjib
fwfgz lilmk uzaed urbwl eitgw
xpbji wfees oubvd dthpk vfmnv
wdnww xczkb wgcdo pvvlp zpfti
ladva scool sshhv lvrtg wrebv
```

In this case there are 25 characters on each line of the one-time pad, and we thus go from [0] through to [24] on the first row, then from [25] to [49] on the second row, and so on. Next if Bob wants to send a message he will select a key based on the positions of the letters:

```
[5][92][4][232][203][70][225][195]
```

If we look up the positions for this, the key becomes:

```
mvvowclv
```

Next we take our secret word, such as “newhampshire”, and shift each letter depending on the position of our key. In this case we translate a “n” to row “m” and get a “z” (which is 12 character shifts ... n[opqrstuvwxy]z:).

```
   a b c d e f g h i j k l m n o p q r s t u v w x y z
. . .
10 k l m n o p q r s t u v w x y z a b c d e f g h i j
11 l m n o p q r s t u v w x y z a b c d e f g h i j k
12 m n o p q r s t u v w x y z a b c d e f g h i j k l
```

The cipher then becomes: “zzrvwoantdms”, which Bob will send to Alice, and Alice does the reverse of Bob’s operation, based on the shared secret key. Unfortunately the OTP cipher suffers from having to regenerate the pad each time, or, at least, to regenerate a new key.

🕮 **Web link (OTP):** http://asecuritysite.com/coding/otp

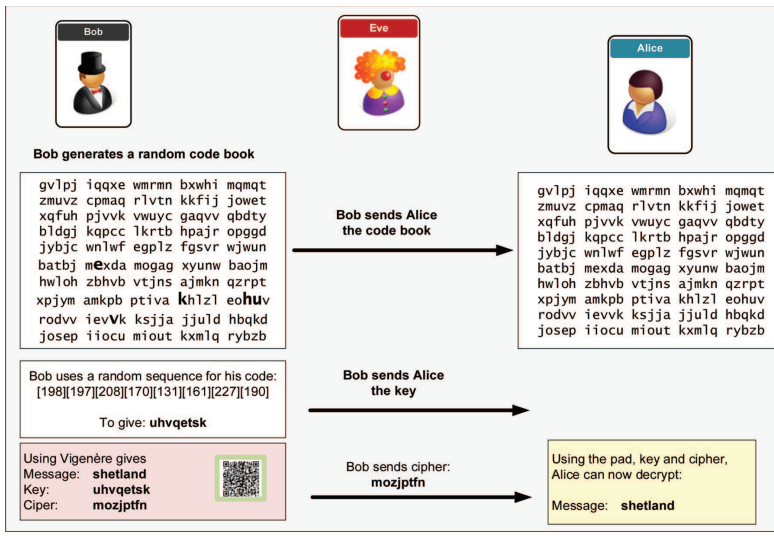


Figure 1.8 One-time pad.

1.3 Encoding Methods

On a computer system, code and data are represented as binary, but humans find it difficult to deal with binary formats, so other formats are used to represent binary values. Two typical formats used to represent characters are ASCII and UTF-16. With ASCII we have 8-bit values and it can thus supports up to 256 different characters (2^8). UTF-16 extends the characters to 16-bit values, and thus gives a total of 65,536 characters (2^{16}). Within ASCII coding, we map printable characters, such as 'a', and 'b', to decimal, binary and hexadecimal values:

ASCII	Binary	Hex	Decimal
'e'	0110 0101	0x65	101
'E'	0100 0101	0x45	69
' '	0010 0000	0x20	32

We also have other 'non-printing' characters which typically have a certain control function. These include CR (Carriage Return), LF (Line Feed), and Horizontal Tab (HT):

ASCII	Binary	Hex	Decimal	Character representation
CR	0110 0101	0x0D	13	\r
LF	0100 0101	0x0A	10	\n
HT	0000 0111	0x07	7	\t

📖 **Web link (ASCII):** http://asecuritysite.com/coding/ascii

Within text files we are likely to have line breaks, and which are created by the CR and LF characters. In Microsoft Windows-type systems, we use CR and LF at the end of a line (\n\r), while a Linux/Mac-type system only uses CR for a new line (\r)

Normally when we encrypt into ciphertext it produces a bit stream which contains non-printing characters, and we thus need to represent the cipher in a printable way. We may also be required to represent our encryption keys in a printable and/or distributable format. For this we often use a hexadecimal or Base-64 format as these allow us to represent the cipher into a printable format (Figure 1.9).

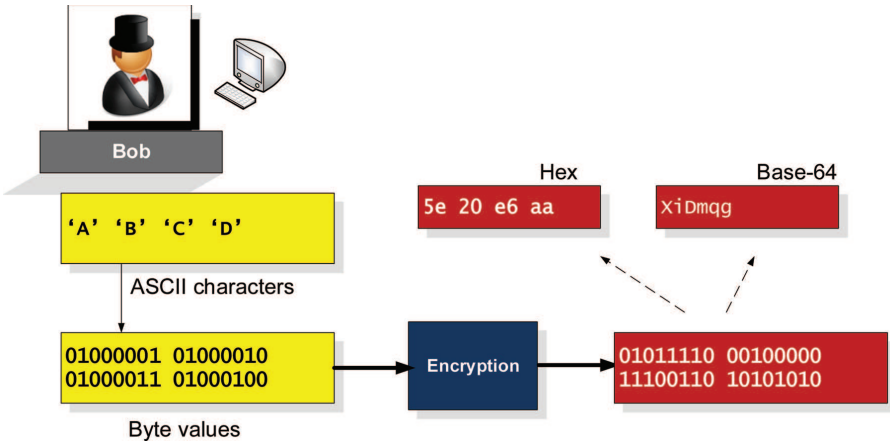


Figure 1.9 Conversion from binary into hexadecimal or Base-64.

The most common format for representing standard English characters is ASCII. In its standard form it uses a 7-bit binary code to represent characters (letters, giving a range of 0 to 127), but it is rather limited in its scope as it does not support symbols such as Greek letters. To increase the number of symbols which can be represented, extended ASCII is used which has a 16-bit code. Appendix A shows the standard ASCII character set (in binary, decimal, hexadecimal and also as a character).

Some important non-printable ASCII characters are: New line (0x13); Carriage Return (0x10); Tab (0x07); and Backspace (0x08), while a Space is represented by 0x20. The representations are for 'A' and 'B' are:

Char	Decimal	UTC-16	ASCII	Hex	Oct	HTML
A	65	00000000 01000001	01000001	41	101	&#65;
B	66	00000000 01000010	01000010	42	102	&#66;

🕮 **Web link (ASCII table):** http://asecuritysite.com/coding/asc
🕮 **Web link (UTF-16 table):** http://asecuritysite.com/coding/asc2
🕮 **Web link (ASCII conversion):** http://asecuritysite.com/coding/ascii

1.3.1 Hexadecimal and Base-64

The conversation to a hexadecimal format involves splitting the bit stream into groups of four bits (Figure 1.10) and for Base-64 into groups of six bits (Figure 1.11). With a hexademical format, we have values from 0 to 15, and

which are represented by four-bit values from 0000 to 1111. For Base-64, we take six bits at a time. For example, if we take an example of "fred", then we get:

ASCII	f	r	e	d
Binary	01100110	01110010	01100101	01100100

To convert to Base-64, we group in 6-bits:

Binary	011001	100111	001001	100101	011001	00

And then map these to a Base-64 table:

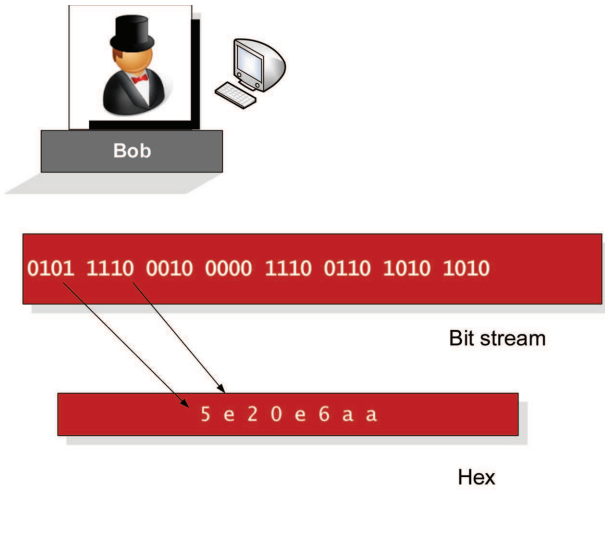
Binary	011001	100111	001001	100101	011001	00
Decimal	25	39	9	37	25	0
Base-64	Z	n	J	l	Z	A

The result is ZnJlZA

With Base-64, we create groups of four Base-64 characters, and we pad with zeros to fill-up the six-bit values, and then use the "=" character to pad to create groups of four Base-64 characters:

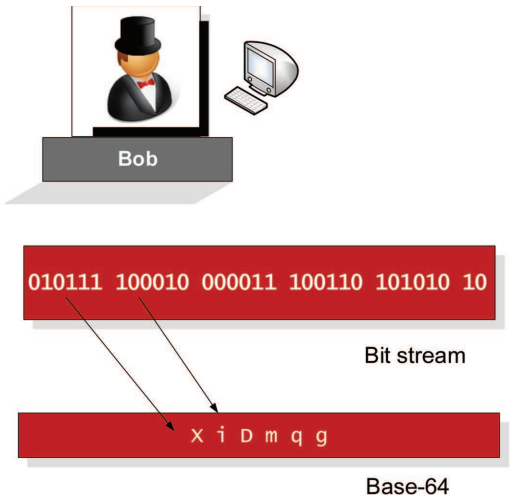
```
test -> 01110100 01100101 01110011 01110100
test -> 011101 000110 010101 110011 011101 00[0000] = =
test ->  d       G      V      z      d    A       = =


help -> 01101000 01100101 01101100 01110000
help -> 011101 000110 010101 110011 011101 00[0000] = =
help ->  a     G       V      s      c    A       = =
```

Unfortunately some of the characters look similar when they are printed, such as whether we have a zero ('0') or an 'O'. To avoid this we can convert to a Base-64 format, but there are similar-looking letters: 0 (zero), O (capital o), I (capital i) and l (lower case L), and non-alphanumeric characters of + (plus) and / (slash). The solution is Base-58, used in Bitcoin applications, and where we remove the characters which are similar looking.



Decimal	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Figure 1.10 Conversion to hex.



Val	Enc	Val	Enc	Val	Enc	Val	Enc
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Figure 1.11 Conversion to Base-64.

For Base-58, we convert the ASCII characters into binary, and then keep dividing by 58 and convert the remainder to a Base58 character. The alphabet becomes:

‘123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz’

It we take an example of ‘e’, where e have a decimal value of 101, so we divide by 58 to get:

1 remainder 43

and next we divide 1 by 58 and we get:

0 remainder 1

We then take character at position 1 and at position 43, to give:

123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

and then get:

2k

If we now take ‘ef’, we get 958 (102 + 101 $\times$ 256), where we move each character up one byte. Basically we take the binary value of the string and then divide by 58 and take the remainder. So ‘ef’ is ‘01100101 01100110’.

📖 **Web link (Base-58 conversion):** http://asecuritysite.com/encryption/base58

1.4 Huffman Coding and Lempel-Viz Welsh (LZW)

Along with encoding methods, we often try to compress our data by either looking at patterns within the binary digits or within the metadata contained in an object. One of the most widely used methods is Huffman Coding which uses a variable length code for each of the elements within the data. This normally involves analyzing the data to determine the probability of its elements, and where the most probable elements are coded with a few bits, and the least probable elements coded with a greater number of bits. This could be done on a character-by-character basis within text-based data, or on a byte-by-byte basis on other binary data (such as for graphics files).

The following example relates to characters. First, the textual data is scanned to determine the number of occurrences of a given letter. For example:

Letter	'b'	'c'	'e'	'i'	'o'	'p'
No. of occurrences:	12	3	57	51	33	20

Next the characters are arranged in order of their number of occurrences, such as:

'e'	'i'	'o'	'p'	'b'	'c'
57	51	33	20	12	3

After this the two least probable characters are assigned either a 0 or a 1. Figure 1.12 shows that the least probable ('c') has been assigned a 0 and the next least probable ('b') has been assigned a 1. The addition of the number of occurrences for these is then taken into the next column and the occurrence values are again arranged in descending order (that is, 57, 51, 33, 20 and 15). As with the first column, the least probable occurrence is assigned a 0 and the next least probable occurrence is assigned a 1. This continues until the last column. When complete, the Huffman-coded values are read from left to right and the bits are listed from right to left.

The final coding will be:

'e' 11
'i' 10
'o' 00
'p' 011
'b' 0101
'c' 0100

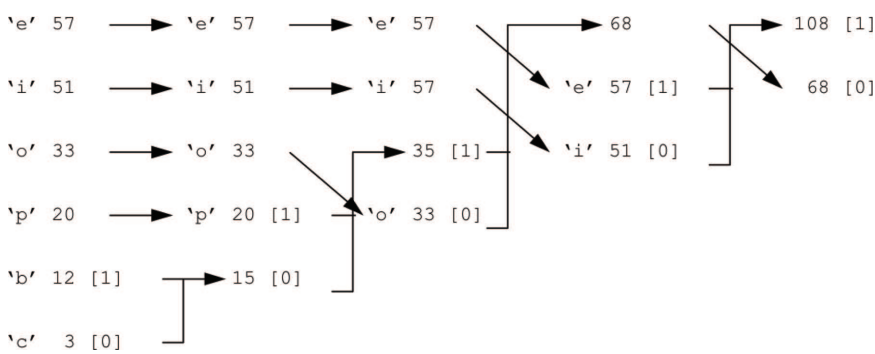


Figure 1.12 Huffman coding example.

📖 **Web link (Huffman):** http://asecuritysite.com/coding/huff

Around 1977, Abraham Lempel and Jacob Ziv developed the Lempel–Ziv class of adaptive dictionary data compression techniques (also known as LZ-77 coding), and which is now the basis of many popular compression methods. The LZ coding scheme is especially suited to data which has a high

degree of repetition, and then makes back-references to these repeated parts. Typically a special flag is used to identify coded and unencoded parts, where the flag creates a back reference to the repeated sequence. An example piece of text could be:

```
'The receiver requires a receipt for it. This is automatically
sent when it is received.'
```

This text has several repeated sequences, such as 'is', 'it', 'en', 're' and 'recei'. For example, we could identify the repetitive sequence of 'recei' (as shown by the underlined highlighted text). If we use an encoded sequence for a flag sequence of *#m#n* then *m* can represents the number of characters to trace back to find the character sequence and *n* is the number of replaced characters. The encoded message would become:

```
'The receiver#9#3quires a#20#5pt for it. This is
automatically sent wh#6#2 it #30#2#47#5ved.'
```

Normally, a long sequence of text has many repeated words and phrases, such as 'and', 'there', and so on. Note that in some cases, this could lead to longer conversions if short sequences were replaced with codes that were longer than the actual sequence itself.

The Lempel–Ziv–Welsh (LZW) algorithm (also known LZ-78) extends LZ-77 by building a dictionary of frequently used groups of characters (or 8-bit binary values), and then rather than storing the actual value, a reference is added to it in a table. Then, before the conversion is decoded, we must read the dictionary. In Figure 1.13 we store the words in a table, and then refer to this in the stored data.

A typical method to then apply to the data is RLE (Run Length Encoding) which takes long sequences of a repeated value and then refers to them in the stored data. For example a sequence of number of:

6,5,5,5,5,5,5,5,5,5,5,10

could become:

6,5 [10],10

where [10] represents ten repeated values.

If we have an input phrase of:

Cows graze in groves on grass which grows in grooves in groves

Then the compressed version could become:

[‘C’, ‘o’, ‘w’, ‘s’, ‘ ’, ‘g’, ‘r’, ‘a’, ‘z’, ‘e’, ‘ ’, ‘i’, ‘n’, 260, ‘r’, ‘o’, ‘v’, ‘e’, 259, ‘o’, 268, 261, ‘a’, ‘s’, 259, ‘w’, ‘h’, ‘i’, ‘c’, ‘h’, 269, 257, 259, 267, 286, 271, 273, 266, 276, 270, 272, ‘s’]

The table would then contain:

Adding: [256] Co
Adding: [257] ow
Adding: [258] ws
Adding: [259] s
Adding: [260] g
Adding: [261] gr
Adding: [262] ra
Adding: [263] az
Adding: [264] ze
Adding: [265] e
Adding: [266] i

📖 **Web link (LZW):** https://asecuritysite.com/comms/lz

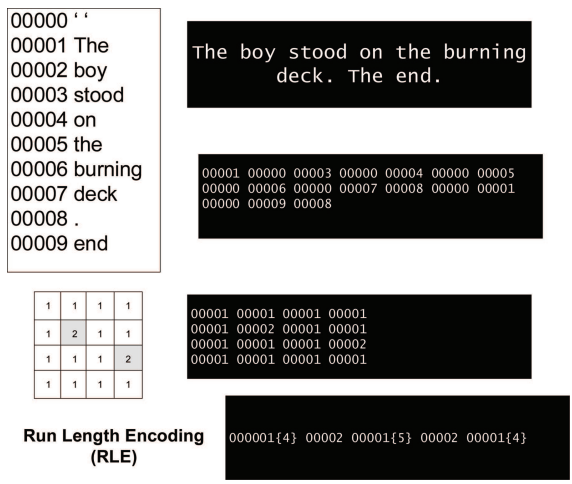


Figure 1.13 LZW and RLE.

1.5 Data Integrity (CRC-32)

Along with keeping things secret, and in proving the identity of an entity, we also need to integrate **integrity**, where we can prove that something has not been changed. A simple method of doing this is to add a checksum, in order to detect errors in the binary digits. CRC (Cyclic Redundancy Check) is one of the most reliable error detection schemes and can detect up to 95.5% of all errors. The most commonly used code is CRC-32 and provides a 32-bit CRC signature (eight hex characters), and which is normally appended onto the data. When data is read, the system checks the CRC-32 value, and if it is different from the expected value there is likely to be an error in the data.

The basic idea of a CRC can be illustrated using an example. Suppose that the Bob and Alice both agree that the numerical value that Bob sends to Alice will always be divisible by 9. Then if Alice receives a value which is not divisible by 9 she knows that the data has an error or has been modified. If the value that Bob is sending is 32, he could multiply the value by 10 to give 320, and then add a value for the least significant digit that would make it divisible by 9. In this case Bob would add 4, making a transmitted value of 324. If this transmitted value were to be corrupted in transmission, there would only be a 10% chance that an error would not be detected. When received without an error, Alice would ignore the least significant digit.

A standard test vector for CRC-32 is "The quick brown fox jumps over the lazy dog" and which generates a CRC-32 value of: 414fa339 (0100 0001 0100 1111 1010 0011 0011 1001).

🕮 **Web link (CRC-32):** http://asecuritysite.com/encryption/crc32

CRC-32 is fairly strong in detecting an error in the transmission, but cannot actually detect the bits what are in error (error correction). More complex schemes exist which can not only detect errors but correct them. One of the most popular schemes is Reed-Solomon:

🕮 **Web link (CRC-32):** http://asecuritysite.com/encryption/reed

1.6 Little Endian or Big Endian

Memories store data in bytes, and where each byte has a unique memory location. The order that the byte values are stored depends on the computer architecture type. Most PC systems using Intel processors use a **Little Endian**

format, where the least significant byte is stored in the lowest memory address. Thus if we have an unsigned 32-bit value of 0x01020304 (16,909,060), the value is stored at:

Location (100h): 01 (Most significant byte)
Location (101h): 02
Location (102h): 03
Location (103h): 04 (Least significant byte – at the end)

Most processors now use the Little Endian format. The Big Endian format has been used in IBM z/Architecture mainframes, where the most significant byte is stored in the lowest memory address. It is also used in network packets such as with TCP and IP headers.

1.7 Introduction to Probability and Number Theory

Encryption requires a background in some basic maths principles, including for the usage of integers and in some basic operations.

1.7.1 Combinations and Permutations

Often we have calculations that involve a number of combinations or permutations. With combinations we do not care the order of the selections. For example if we have four countries: UK, France, Germany and Ireland, there are four combinations of three countries:

[{UK, France, Germany}, {UK, France, Ireland}, {France, Germany, Ireland} and {UK, Germany, Ireland}]

The formula for this is:

$$ {}_nC_k = \frac{n!}{k!(n-k)!} $$

Where ! is the factorial operator, *n* is the total to choose from, and *k* is the number of options to choose. For example 5! is 5×4×3×2×1. So for our example we get:

$$ {}_4C_3 = \frac{4!}{3!(4-3)!} = \frac{4 \times 3 \times 2 \times 1}{3 \times 2 \times 1} = 4 $$

With permutations we look at all the options including their sequence. For example, the number of permutations for three countries with our example is:

[{UK, France, Germany}, {UK, France, Ireland}, {UK, Germany, France}, {UK, Germany, Ireland}, {UK, Ireland, France}, {UK, Ireland, Germany}, ... {France, Germany, Ireland}]

The formula for this is:

$$ {}_nP_k = \frac{n!}{(n-k)!} $$

So for our example we get:

$$ {}_4P_3 = \frac{4!}{(4-3)!} = \frac{4 \times 3 \times 2 \times 1}{1} = 24 $$

1.7.2 Probability Theory

With probability theory we determine the likelihood of an event happening, typically by understanding the chances of how each of the elements involved in an event interact, and the likelihood of them happening. For a dice, each of the numbers of a dice are equally likely, thus the probability of us rolling a specific value of *n* is:

$$ \mathrm{P}(n) = \frac{1}{6} \approx 0.167 $$

If we have two events (A and B) that are **independent**, the probability of both occurring is:

$$ \mathrm{P}(\mathrm{A \text{ and } B}) = \mathrm{P}(\mathrm{A}).\mathrm{P}(\mathrm{B}) $$

If the events are **mutuality exclusive**, such as, if we toss a coin, and if it is heads, it cannot also be tails. So for mutuality exclusive events:

$$ \mathrm{P}(\mathrm{A \text{ and } B}) \; = \; 0 $$

If we want to determine the probability of one of two events (A or B) happening:

$$ \mathrm{P}(\mathrm{A \text{ or } B}) \; = \; \mathrm{P}(\mathrm{A}) + \mathrm{P}(\mathrm{B}) $$

For example, the probability of rolling a two or a three on dice, the probability will be:

$$ \mathrm{P}(2 \text{ or } 3) \; = \; 1/6 \; + \; 1/6 \; = \; 1/3 $$

If we throw twice dice, each of the die are independent, so the chances of two 1's being thrown is 1/6 multiplied by 1/6, which equals 1/36.

If the events are **dependent**, we need to understand the dependence. For example what is the probability of drawing two Aces from a pack of 52 cards? The first pick has a probability of 4/52 (1/13), but if we selected an Ace first, then there will only be 3 Aces left out of 51 cards, so the chances of drawing another Ace will be 3/51. Overall the probably of drawing two Aces will thus be:

$$P(A \text{ and } A) = \frac{1}{13}\frac{3}{51} = \frac{3}{663} = 0.0045$$

In general, for two dependent events (A and B), we have:

$$\mathrm{P(A \text{ and } B)} \;=\; \mathrm{P(A)} \cdot \mathrm{P(B|A)}$$

Where P(B|A) is the probability of B happening given that A happened.

1.7.3 Set Theory

With set theory we define a range objects that make up a set. If we create two sets named Players and Spectators:

Players — {mike, fred, bert}
Spectators — {ian, michael, mike}

The main symbols that we use are:

Symbol	Symbol Name	Description
|	such that	so that
A$\bigcap$B	intersection	objects belong to set A **and** set B
A$\bigcup$B	union	objects belong to set A **or** set B
A$\subseteq$B	subset	subset has fewer elements or equal to the set
$\in$	belongs to	when an object is within a set
$\notin$	does not belong to	when an object is not in a set

Thus A$\bigcap$B — {mike} and A$\bigcup$B —{mike,fred,bert,ian,michael}.
Then 'mike' $\in$ Players, and 'ian' $\notin$ Players.

1.7.4 Number Representations

Many of the important concepts in cryptography are based on number theory around the study of integers, with a special focus on divisibility. The main classifications for numbers are: integers; rational numbers; real numbers; and complex numbers. In maths we define these as:

- Integers can be positive or negative numbers and have no fractional part. They are represented with the $\mathbb{Z}$ symbol $\{\ldots$-2, -1, 0, +1, +2,$\ldots\}$.
- Rational numbers are fractions ($\mathbb{Q}$).
- Real numbers ($\mathfrak{R}$) include both integers and rational numbers, and any other number that can be used in a comparison.
- Prime numbers ($\mathbb{P}$) represent the integers which can only be divisible by itself and unity.
- Natural numbers ($\mathbb{N}$) represent positive numbers which are integers $\{1,2\ldots\}$.

1.7.5 Logarithms

There are some methods in cryptography which base themselves on logarithms. They were discovered by John Napier, and who first proposed that we can multiply two numbers together (a, b), by finding the finding the log of a and adding it to the log of b. We can then take the inverse log to determine the result. This changed the face of calculations, where we could multiply large numbers together, just by looking up a table for the log value, and adding the results, and again apply the reverse through a look-up table. To multiply we get:

$$a \times b = \text{Inverse Log } (\text{Log } (a) + \text{Log } (b))$$

The base of the log is important for the calculation. For our decimal system we use a base of 10 ($\log_{10}$(x) and 10^x), but for many mathematical operations we use a natural log base (Log_e(x) or e^x, where e has a value of approximately 2.718). The base of e is used in many naturally occurring changes, such as within electrical circuits. The rules are thus:

$g = a.b$
$\log(g) = \log(a)+\log(b)$
g = Inverse Log ($\log(a)+\log(b)$)

$g = a/b$
$\log(g) = \log(a)-\log(b)$
g = Inverse Log ($\log(a)-\log(b)$)

$g = a^x$
$\log(g) = x.\log(a)$
g = Inverse Log ($x.\log(a)$)

For example:

$g = 10^3$
$\log_{10}(g) = 3.\log_{10}(10)$
$g = 10^{(3\times1)} = 1{,}000$

1.8 Prime Numbers

A prime number is a value which only has factors of 1 and itself, and are used in areas such as key exchange and in public key encryption. Their core protection is that it is a significant challenge for a computer to factorise the result of the multiplication of two prime numbers. The simplest test for a prime number is to divide the value from all the integers from 2 to the value divided by 2. If any of the results leaves no remainder, the value is not prime, otherwise it is composite. We can obviously improve on this by getting rid of even numbers which are greater than 2, and also that the highest value to be tested is the square root of the value.

So if $n = 37$, then our maximum value will be $\sqrt{n}$, which, when rounded down is 6. So we can try: 2, 3, and 5, of which of none of these divide exactly into 37, so it is a prime number. Now let us try 55, where we will then try 2, 3, 5 and 7. In this case 5 does divide exactly into 55, so the value is not prime.

Another improvement we can make is that prime numbers (apart from 2 and 3) fit into the equation of:

$$6k \pm 1$$

where k=0 gives 0 and 1, k=1 gives 5 and 7, k=2 gives 11 and 13, k=3 gives 17 and 19, and so on. Thus we can test if we can divide by 2 and then by 3, and then check all the numbers of $6k \pm 1$ up to $\sqrt{n}$.

🕮 **Web link (Prime Numbers):** http://asecuritysite.com/encryption/isprime

1.9 Encryption Operators (mod, EX-OR and shift)

It is important that the operators used in encryption do not lose any information in the encryption process, and that the operations must be reversible in some way. Along with this, the encryption process is often fairly

processor-intensive, so the operators must be fairly simple in their approach in order to be fast for Bob and Alice, but which involves extensive processing for Eve. The main operators which match best to this profile are: bit rotate ($<<$ or $>>$); eXclusive-OR (**X-OR** - $\oplus$); and **mod** operations. These can typically be achieved in a single operation, and can thus be used for fast encryption and decryption. Along with this the rotate and X-OR functions are fairly easy to reverse.

1.9.1 Mod Operator

The **mod** operator provides the remainder of an integer divide. For example for 31 divided by 8 gives the result of 3 remainder 7. Thus 31 (**mod** 8) equals 7. Often in cryptography the mod operation uses a prime number, such as:

$$\text{Result} \;=\; \text{value}^{x} \textbf{ mod } (\text{prime number})$$

For example, if we have a prime number of 269, and a value of 8 with an x value of 5, the result of this operation will be:

$$\text{Result} \;=\; 8^{5} (\textbf{mod } 269) \;=\; 32,768 \; (\textbf{mod } 269) \;=\; 219$$

With prime numbers, if we know the result, it is difficult to find the value of x that has been used, even though we have the other values, as there can be many values of x that can produce the same result. It is this feature which makes it difficult to determine a secret value (in this case the secret is x). In Python, Java and C#, the mod operator is "%".

1.9.2 Shift-Operators

The bit-shift operators can either be left- or right-shift (or more precisely rotate left, or rotate right operators), where the shifting process normally takes the bits which exit from one end, and put them into the other end. This is normally defined as a rotation – where we can have a rotate left or a rotate right. An encryption process might thus operate by taking one byte at a time and rotate them left by four bit positions:

Input	1010	1000	1111	0000	0101	1100	0000	0001
Output	1000	1010	0000	1111	1100	0101	0001	0000

Thus the decryption process would merely rotate each of the bits of the bytes by four places to the right.

1.9.3 Integers and Big Integers

In computer systems we represent integers with a number of bits. Normally in cryptography we use unsigned integers in order to apply simple operations on the values. A typical integer representations are:

C# data type	**Representation**	**Range**
byte	byte	uses 8 bits and ranges from 0 to 255
ushort	unsigned short	uses 16 bits and ranges from 0 to 65,535
uint	unsigned int	uses 32 bits and ranges from 0 to 42,949,67,295
ulong	unsigned long	uses 64 bits and ranges from 0 to 18,446,744,073,709,551,615

Thus when we use the shift operator, the variables are automatically cast against their variable types. In C the "<<" operator shifts left, and the ">>" operator shifts right. Unfortunately in most software development languages there is no rotate operator, so the bits which move off the end are required to be pushed back onto the other end. In C a function to produce a rotate right for a variable (v) by n bits is:

```
var ror(var v,unsigned int b) {
     return (v>>n)|(v<<(8*sizeof(var)-n));
  }
```

The number of bits used to define an integer is often defined by the size of the registers which are used in the processor. In most cases the maximum size is 64 bits – and which is represented by an unsigned long value (ulong). In cryptography we often have values which are much greater than this and where our integer values can have 2,048 bits or more. Thus the normal data integer types will not support these operators, and there can be an overflow within the operations. We thus use Big Integers to perform the operations, and which store their values as string entities, and not as numeric values. This can thus support almost any number that we need to generate. When the values are operated on, the strings are converted into a numerical format, and the operations performed, and the result placed back into a string format.

A popular implement of Big Integers is the Bouncy Castle library, where, in C#, the following calculates 2 to the power of a given number (*i*):

```
BigInteger b = new BigInteger(''2'');
BigInteger c = b.Pow(i);
```

📖 **Web link (Big Integers):** http://asecuritysite.com/encryption/keys3

The values are then declared as Big Integers objects and can be displayed by converting to a string. For example, if we want to calculate:

$$A = g^x \bmod (n)$$
$$B = g^y \bmod (n)$$

$$k_1 = B^x \bmod (n)$$
$$k_2 = A^y \bmod (n)$$

we can implement the following (where *x* and *y* are random values between 0 and 90, and *g* and *n* are constant values):

```
int x = Global.random(90);
int y = Global.random(90);

BigInteger g = new BigInteger("153d5d6172adb4cb9a428cc", 16);
BigInteger n = new BigInteger("9494fec095f3b8ca98cdf3b", 16);

BigInteger A = g.Pow(x).Mod(n);
BigInteger B = g.Pow(y).Mod(n);

BigInteger k1=B.Pow(x).Mod(n);
BigInteger k2=A.Pow(y).Mod(n);

String k1value = g.ToString();
String k2value = n.ToString();
```

📖 **Web link (Example):** http://asecuritysite.com/encryption/diffie2

The following defines the maximum value that can be represented for various integer bit sizes:

Int size	Number of values
16	**65,536**
32	**4,294,967,296**
48	281,474,976,710,656
64	**18,446,744,073,709,551,616**
80	1,208,925,819,614,629,174,706,176
96	79,228,162,514,264,337,593,543,950,336
112	5,192,296,858,534,827,628,530,496,329,220,096
128	340,282,366,920,938,463,463,374,607,431,768,211,456
144	22,300,745,198,530,623,141,535,718,272,648,361,505,980,416
160	1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,97 6
176	95,780,971,304,118,053,647,396,689,196,894,323,976,171,195,136,4 75,136
192	6,277,101,735,386,680,763,835,789,423,207,666,416,102,355,444,46 4,034,512,896
208	411,376,139,330,301,510,538,742,295,639,337,626,245,683,966,408, 394,965,837,152,256
224	26,959,946,667,150,639,794,667,015,087,019,630,673,637,144,422,5 40,572,481,103,610,249,216
240	1,766,847,064,778,384,329,583,297,500,742,918,515,827,483,896,87 5,618,958,121,606,201,292, 619,776

1.9.4 X-OR

Along with the shift operators another important operator is the bitwise X-OR operation ($\oplus$). Its basic function is:

Bit1	**Bit2**	**Output**
0	0	0
1	0	1
0	1	1
1	1	0

An example of an operation with an X-OR of 0101 0101 for each byte is:

Input	1010 1000	1111 0000	0101 1100	0000 0001
X-OR	0101 0101	0101 0101	0101 0101	0101 0101
Output	**1111 1101**	**1010 0101**	**0100 1001**	**0101 0100**

The great advantage of the X-OR bitwise operation is that, like the bit rotate operators, it preserves the information in the processed output, and can be undone by merely operating on the output with the same value that was used to generate the result. For example:

Output	1111 1101	1010 0101	0100 1001	0101 0100	
X-OR	0101 0101	0101 0101	0101 0101	0101 0101	
Input	**1010 1000**	**1111 0000**	**0101 1100**	**0000 0001**	Same value

The following shows an example conversion, where we have a string (“Test”) and apply a key, with an resulting encoded format of “IBEHAA==”:

	ASCII	**Hex**	**Hex (Base-64)**	**Hex (Binary)**
	(Result)	**Hex**		
Input	Test	54657374	VGVzdA==	01010100 01100101 01110011 01110100
Key				01110100 01110100 01110100 01110100
Encoded		20110700	IBEHAA==	00100000 00010001 00000111 00000000

A simple encryption process might be:

- Take 32 bits at a time.
- Shift bits by four spaces to the left.
- X-OR the value by 1010 1000.
- Shift bits by two spaces to the right.
- X-OR the value by 1010 1000.

Then, the decryption process would be (reading 32 bits at a time):

- X-OR the value by 1010 1000
- Shift bits by two spaces to the left.
- X-OR the value by 1010 1000.
- Shift bits by four spaces to the right.

1.9.5 Modulo-2 Operations

In cryptography we try and avoid complex mathematical operations which involve carry-overs for bits. This type of operation is known as Modulo-2, or GF(2) – which is a Galois field of two elements – and is used in many areas including with checksums and ciphers. The multiplication function involves multiplying the binary values and ignoring the remainder from each carry forward. This type of operation simplifies the implementation and is fast in its operation. It basically involves some bit shifts and an EX-OR function, and which makes it fast in computing the multiplication.

The basic operations are:

0+0=0 1+1=0
0+1=1 1+0=1

It performs the equivalent operation to an exclusive-OR (XOR) function. For modulo-2 arithmetic, subtraction is the same operation as addition:

0–0=0	1–1=0
0–1=1	1–0=1

Multiplication is performed with the following:

$0\times0=0$	$0\times1=0$
$1\times0=0$	$1\times1=1$

which is an equivalent operation to a logical AND operation.

Binary digit representation, such as 101110, is often difficult to use when multiplying and dividing, so a typical representation is to manipulate the binary value as a polynomial of bit powers. This technique represents each bit as an x to the power of the bit position and then adds each of the bits, such as:

10111	x^4+x^2+x+1
1000 0001	x^7+1
1111 1111 1111 1111	$x^{11}+x^{10}+x^9+x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1$
10101010	$x^7+x^5+x^3+x$

For example:	101×110
is represented as:	$(x^2+1)\times(x^2+x)$
which equates to:	$x^4+x^3+x^2+x$
which is thus:	11110

📖 **Web link (Example):** http://asecuritysite.com/calculators/mod2

The addition of the bits is treated as a Modulo-2 addition, where any two variables which have the same powers are equal to zero (1+1=0). For example:

$$x^4 + x^4 + x^2 + 1 + 1$$

is equal to x^2 as x^4+x^4 equates to zero and 1+1 equates to 0 (in modulo-2). An example which shows this is the multiplication of 10101 by 01100.

Thus:	10101×01110
is represented as:	$(x^4+x^2+1)\times(x^3+x^2+x)$
which equates to:	$x^7+x^6+x^5+x^5+x^4+x^3+x^3+x^2+x$
which equates to:	$x^7+x^6+x^4+x^2+x$
which is thus:	11010110

The division process uses an exclusive-OR operation instead of subtraction and can be implemented with a shift register and a few XOR gates. For example, 101101 divided by 101 is implemented as follows:

```
      1011
100| 101101
     100
       110
       100
        101
        100
          1
```

Thus, the modulo-2 division of 101101 by 100 is thus 1011 remainder 1. As with multiplication, this modulo-2 division can also be represented with polynomial values.

📖 **Web link (Example):** http://asecuritysite.com/comms/mod_div

1.10 GCD

GCD is known as the greatest common divisor, or greatest common factor (gcf), and is the largest positive integer that divides into two numbers without a remainder. For example, the GCD of 9 and 15 is 3. It is an operation that is used many encryption algorithms, and example of some code to calculate the GCD for two values (a and b) is:

```
static int GCD(int a, int b)
{
    int Remainder;
    while( b != 0 )
    {
        Remainder = a % b;
        a = b;
        b = Remainder;
    }
    return a;
}
```

If we run with a value of 54 and 8, we get:

a:54, b:8, Remainder:6
a:8, b:6, Remainder:2
a:6, b:2, Remainder:0
Return value:2

📖 **Web link (Example):** http://asecuritysite.com/encryption/gcd

1.11 Random Number Generators

Within cryptography random numbers are used to generate things like encryption keys. If the generation of these keys can be predicted in some way, it may be possible to guess them. The two main types of random number generators are:

- **Pseudo-Random Number Generators** (PRNGs). This method repeats the random numbers after a given time (periodic). They are fast and are also deterministic, and are useful in producing a repeatable set of random numbers.
- **True Random Number Generators** (TRNGs). This method generates a true random number, and uses some form of random process. One approach is to monitor the movements of a mouse pointer on a screen or from the pauses between keystrokes. Overall the method is generally slow, especially if it involves human interaction, but is non-deterministic and aperiodic.

Normally simulation and modelling applications use PRNG, so that the values generated can be repeated for different runs, while cryptography, lotteries, gambling and games use TRNG, as each value should not repeat or be predictable. If the generation of key was deterministic, Eve could possibly guess the key created. So, in the generation of encryption keys for public key encryption, users are often asked to generate some random activity, and where a random number is then generated based on this activity. This random number is then used to generate the encryption keys.

Computer programs, though, often struggle to generate truly random numbers, so hardware generators are often used within highly secure applications. One method is to generate a random number based on low-level, statistically random *noise* signals. This includes things like thermal noise and from the photoelectric effect.

🕮 **Web link (Random number):** http://asecuritysite.com/encryption/random

1.11.1 Linear Congruential Random Numbers

One method of creating a simple random number generator is to use a sequence generator of the form:

$$X_{i+1} \leftarrow (a \times X_i + c) \bmod m$$

Where *a*, *c* and *m* are integers, and where X_0 is the seed value of the series. If we take the values of *a*=21, X_0=35, *c*=31 and *m*=100 we get a series of:

(21×35+31) mod 100 gives 66
(21×66+31) mod 100 gives 17
(21×17+31) mod 100 gives 88
and so on.

```
66 17 88 79 90 21 72 43 34 45 76 27 98 89 0 31 82 53
```

📖 **Web link (Linear congruential):** http://asecuritysite.com/encryption/linear

Within cryptography, it is important that we are generating values which are as near random as possible, so that Eve cannot guess the random numbers that Bob and Alice have used. With randomness we cannot determine how random the values are, by just taking a few samples. For this we need a large number of samples, and take an estimation of the overall randomness.

There are various tests for randomness. For example, we could define an average value which is half way between the number range, and then determine the ratio of the values above and below the half way point. This will work, but will not show us if the values are well distributed. Along with this we could determine the arithmetic mean of the values, and match it to the centre value within the range of numbers.

An improved method to test for the distribution of values is the Monte Carlo value for Pi test. With this method, we take our random numbers and scale them between –1.0 (scaled from the minimum value) and 1.0 (scaled from the maximum value). Next we take two values at a time and calculate:

$$\sqrt{x^2 + y^2}$$

If this value is less than or equal to one, we place in the circle (with a radius of 1), otherwise it is out of the circle. The estimation of PI is then four times the number of points in the circle (M) divided by the total number of points (N). In Figure 1.14, the blue points are outside the circle and the yellow ones are inside.

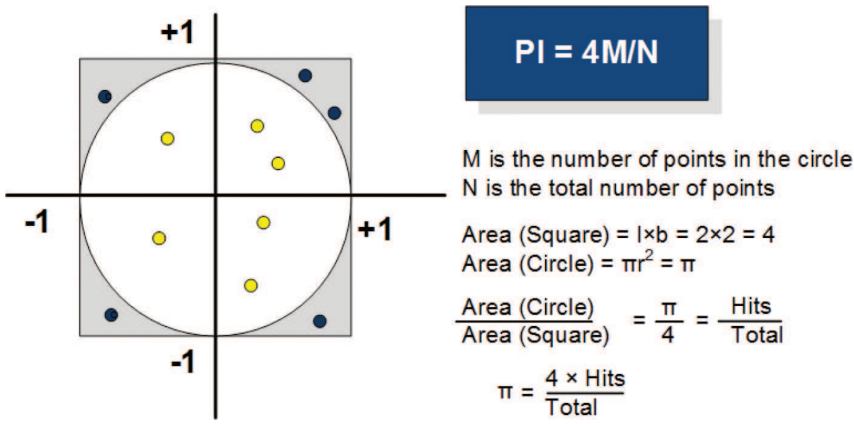


Figure 1.14 Analysis of cipher text compared with normal probabilities.

🕮 **Web link (Monte Carlo):** http://asecuritysite.com/encryption/mc

Another method for determining randomness is to measure the entropy of the data. Entropy was defined by Claude E. Shannon in his 1948 paper, and where the maximum entropy occurs when there is an equal distribution of all bytes across the data. Normally we define these in terms of bytes. The method we use is to take the frequencies of the byte values and calculate how many bits are used:

$$E_n = \sum_{n=1}^{n=255} f_n \log_2(f_n)$$

where f_n relates to the frequency of the byte values. A maximum entropy is 8 bits (for a byte value). For values from 0 to 255, we would expect a result around 8 bits if the values are random.

🕮 **Web link (Entropy):** http://asecuritysite.com/encryption/ent

1.12 Frequency Analysis

Finally, we will do a little bit of frequency analysis, as it is often used in cipher cracking, especially to spot variations in the probably of codes. It is best illustrated with an example. If our cipher text is:

```
LQ A EAOONM WC A CNI UNHAUNZ OKN IWMRU KAZ HKAQDNU CMWE AQ
LQUSZOMLAR ADN LQOW AQ LQCWMEAOLWQ ADN. LO LZ WQN IKLHK,
SQRLPN NAMRLNM ADNZ, NQHATZSRAONZ FLMOSARRB OKN IKWRN IWMRU.
LO LZ ARZW WQN IKLHK ARRWIZ OKN QNI LQUSZOMLNZ OW XN XAZNU
LQ AQB RWHAOLWQ ILOKWSO MNVSLMLQD AQB QAOSMAR MNZWSMHNZ,
WM OW XN LQ AQB AHOSAR TKBZLHAR RWHAOLWQZ. OBTLHARRB ARR OKAO
LZ MNVSLMNU LZ A MNRLAXRN QNOIWMP HWQQNHOLWQ. WSM IWMRU LZ
HKAQDLQD XB OKN UAB, AZ OMAULOLWQAR CWMEZ WC XSZLQNZZ AMN
XNLQD MNTRAHNU, LQ EAQB HAZNZ, XB EWMN MNRLAXRN AQU CAZONM
IABZ WC WTNMAOLQD. WSM TWZOAR ZBZONE, IKLRN ZOLRR SZNU
CWM EAQB SZNCSR ATTRLHAOLWQZ, KAZ XNNQ RAMDNRB MNTRAHNU XB
NRNHOMWQLH EALR. ILOK FWOLQD, OKN ZRWI AQU HSEXNMZWEN OAZP WC
EAMPLQD FWOLQD TA-TNMZ ILOK OKN TMNCNMMNU HAQULUAON, LZ
QWI XNLQD MNTRAHNU XB NRNHOMWQLH FWOLQD. OKN OMAULOLWQAR
ZBZONEZ,OKWSDK, KAFN XNNQ AMWSQU CWM KSQUMNUZ LC QWO
OKWSZAQUZ WC BNAMZ, AQU OBTLHARRB SZN INRR OMLNU-AQU-ONZONU
ENHKAQLZEZ. CWM OKN EWZO TAMO, CWM NJAETRN, IN OMSZO
A TATNM-XAZNU FWOLQD ZBZONE, NFNQ OKWSDK LO LZ INRR PQWIQ
OKAO A HWSQO WC OKN FWONZ ILOKLQ AQ NRNHOLWQ ILRR WCONQ
TMWUSHN ULCCNMNQO MNZSROZ NAHK OLEN OKAO OKN FWON LZ HWSQONU,
AQU OKNQ MNHWSQONU. AQ NRNHOMWQLH ENOKWU ILRR, WQ OKN
WOKNM KAQU, EWZO RLPNRB KAFN A ZSHHNZZ MAON WC 100%.
```

We can now determine the frequency of the characters:

a	b	c	d	e	f	g	h	i	j	k	l	m
90 [8.5%]	23 [2.2%]	22 [2.1%]	18 [1.7%]	21 [2.0%]	10 [0.9%]	0 [0.0%]	37 [3.5%]	24 [2.3%]	1 [0.1%]	43 [4.1%]	75 [7.1%]	61 [5.8%]
n	o	p	q	r	s	t	u	v	w	x	y	z
119 [11.2%]	90 [8.5%]	6 [0.6%]	76 [7.2%]	59 [5.6%]	31 [2.9%]	19 [1.8%]	38 [3.6%]	2 [0.2%]	73 [6.9%]	16 [1.5%]	0 [0.0%]	67 [6.3%]

And we can plot the occurrence against how the characters relate to standard English character probabilities (Figure 1.15).

The following table shows how the text matches the normal probability to the text (where ‘E’ has the highest level of occurrence and ‘Z’ has the least). The rows with lower case show what would be expected for the order, and the upper case ones shows what the cipher text gives for the order:

e	t	a	o	i	n	s	h	r	d	l	c	u
N	O	A	Q	L	W	Z	M	R	K	U	H	S
m	w	f	g	y	p	b	v	k	x	j	q	z
I	B	C	E	T	D	X	F	P	V	J	G	Y

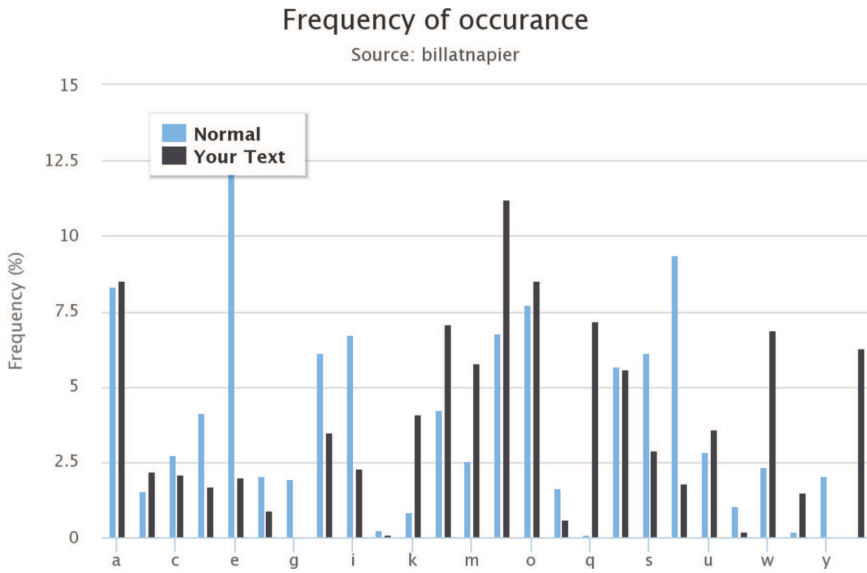


Figure 1.15 Analysis of cipher text compared with normal probabilities.

From this we predict:

- That N in the cipher text maps to e in plaintext.

The next best guess is to analyse the one, two and three-letter characters:

One letter (Most pop: a, I)	**Two letter** (Most pop: of, to, in, it, is, be, as, at, so, we, he, by, or, on, do, if, me, my, up, an, go, no, us, am)	**Three letter** (Most Pop: the, and, for, are, but, not, you, all, any, can, had, her, was, one, our, out, day, get, has, him, his, how, man, new, now, old, see, two, way, who, boy, did, its, let, put, say, she, too, use)
a [90]	lq [20] wc [8] aq [20] lo [9] lz [9] ow [3] xn [7] wm [12] xb [4] az [8] lc [2] in [3] wq [15]	cni [1] okn [13] kaz [2] adn [3] wqn [2] qni [1] aqb [5] arr [5] wsm [3] amn [1] aqu [8] cwm [6] qwi [2] qwo [1] szn [3]

Probably the best guess is that an ‘A’ is an ‘a’, and we can locate “the” with “OKN”, which is the most popular for three letter words. Thus we have ‘O’ mapped to ‘t’, ‘K’ to ‘h’. If we have mapped ‘T’ to ‘a’, then “AQU” looks like it is “and”, and thus gives us ‘Q’ maps to ‘n’ and ‘U’ maps to ‘d’. This gives:

```
Ln a EatteM WC a CeI deHadeZ the IWMRd haZ HhanDed CMWE an
LndSZtMLaR aDe LntW an LnCWMEatLWn aDe. Lt LZ Wne IhLHh,
SnRLPe eaMRLeM aDeZ, enHaTZSRateZ FLMtSaRRB the IhWRe
IWMRd. Lt LZ aRZW Wne IhLHh aRRWIZ the neI LndSZtMLeZ tW Xe
XaZed Ln anB RWHatLWn ILthWSt MeVSLMLnD anB natSMaR MeZWSMHeZ,
WM tW Xe Ln anB aHtSaR ThBZLHaR RWHatLWnZ. tBTLHaRRB aRR that
LZ MeVSLMed LZ a MeRLaXRe netIWMP HWnneHtLWn. WSM IWMRd LZ
HhanDLnD XB the daB, aZ tMadLtLWnaR CWMEZ WC XSZLneZZ aMe
XeLnD MeTRaHed, Ln EanB HaZeZ, XB EWMe MeRLaXRe and CaZteM
IaBZ WC WTeMatLnD. WSM TWZtaR ZBZteE, IhLRe ZtLRR SZed CWM
EanB SZeCSR aTTRLHatLWnZ, haZ Xeen RaMDeRB MeTRaHed XB
eReHtMWnLH EaLR. ILth FWtLnD, the ZRWI and HSEXeMZWEe taZP
WC EaMPLnD FWtLnD Ta-TeMZ ILth the TMeCeMMed HandLdate, LZ
nWI XeLnD MeTRaHed XB eReHtMWnLH FWtLnD. the tMadLtLWnaR
ZBZteEZ, thWSDh, haFe Xeen aMWSnd CWM hSndMedZ LC nWt
thWSZandZ WC BeaMZ, and tBTLHaRRB SZe IeRR tMLed-and-teZted
EeHhanLZEZ. CWM the EWZt TaMt, CWM eJaETRe, Ie tMSZt a
TaTeM-XaZed FWtLnD ZBZteE, eFen thWSDh Lt LZ IeRR PnWIn
that a HWSnt WC the FWteZ ILthLn an eReHtLWn ILRR WCten
TMWdSHe dLCCeMent MeZSRtZ eaHh tLEe that the FWte LZ
HWSnted, and then MeHWSnted. an eReHtMWnLH EethWd ILRR,
Wn the WtheM hand, EWZt RLPeRB haFe a ZSHHeZZ Mate WC 100%.
```

It now becomes easy by scanning an eye over it, where "nWt" looks like 'W' is an 'o', and "Xe" looking like "be", so 'X' becomes a 'b', "Ln" looks like it is "In", which makes a 'L' mapping to 'i'. If we just look at the first line we get:

```
in a EatteM oC a CeI deHadeZ the IoMRd haZ HhanDed CMoE an
indSZtMiaR aDe into an inCoMEation aDe. it iZ one IhiHh,
SnRiPe eaMRieM aDeZ, enHaTZSRateZ FiMtSaRRB the IhoRe IoMRd.
it iZ aRZo one IhiHh aRRoIZ the neI indSZtMieZ to be baZed
in anB RoHation IithoSt MeVSiMinD anB natSMaR MeZoSMHeZ,
oM to be in anB aHtSaR ThBZiHaR RoHationZ.
```

And we can spot ... "EatteM" maps to "matter" and "oC" to "of", "inCoMEation" maps to "information":

```
in a matter of a feI deHadeZ the IorRd haZ HhanDed from an
indSZtriaR aDe into an information aDe. it iZ one IhiHh,
SnRiPe earRier aDeZ, enHaTZSRateZ FirtSaRRB the IhoRe IorRd.
it iZ aRZo one IhiHh aRRoIZ the neI indSZtrieZ to be baZed
in anB RoHation IithoSt reVSirinD anB natSraR reZoSrHeZ,
or to be in anB aHtSaR ThBZiHaR RoHationZ.
```

"feI" then maps to "few", "worRd" to "world", "indSZtriaR" to "industrial" to give:

```
in a matter of a few deHades the world has HhanDed from an
industrial aDe into an information aDe. it is one whiHh,
unliPe earlier aDes, enHaTsulates FirtuallB the whole world.
```

And it doesn't take too many guesses to end up with:

```
in a matter of a few decades the World has changed from
an industrial age into an information age. it is one which,
unlike earlier ages, encapsulates virtually the whole World.
```

📖 **Web link (Frequency Analysis):** http://asecuritysite.com/coding/freq
📖 **Web link (Challenge):** http://asecuritysite.com/challenges/scramb

1.13 Lab/tutorial

The lab and tutorial related to this chapter is available on-line at:

http://asecuritysite.com/crypto01

A few cipher challenge is available at:

http://asecuritysite.com/challenges