

3 Encryption

☐ <http://buchananweb.co.uk/security00.aspx>, Select **Principles of Encryption**.

3.1 Introduction

The key objectives of this unit are to:

- Define the methods used in encryption, especially for public and private key encryption.
- Understand methods that can be used to crack encrypted content.
- Outline a range of standard encryption methods.

3.2 Introduction

The future of the Internet, especially in expanding the range of applications, involves a much deeper degree of privacy, and authentication. Without these the Internet cannot be properly used to replace existing applications such as in voting, finance, and so on. The future is thus towards data encryption which is the science of cryptography¹, and provides a mechanism for two entities to communicate without any other entity being able to read their messages. In a secret communications system, Bob and Alice should be able to communicate securely, without Eve finding out the contents of their messages, or in keeping other details secure, such as their location, or the date that their messages are sent (Figure 3.1).

The two main methods used are to either use a unique algorithm which both Bob and Alice know, and do not tell Eve, or they use a well-known algorithm, which Eve also knows, and use some special electronic key to uniquely define how the message is converted into ciphertext, and back again. A particular problem in any type of encryption is the passing of the secret algorithm or the key in a secure way, as Bob or Alice does not know if Eve is listening to their communications. If Eve finds-out the algorithm or the key, neither Bob nor Alice is able to detect this. This chapter looks at some of the basic principles of encryption, including the usage of private-key and public-key methods. As we will find public and private key methods work together in perfect harmony, with, typically, private key methods providing in the actual core encryption, and public key methods providing ways to authenticate, and pass keys.

¹ The word *cryptography* is derived from the Greek words which means hidden, or secret, writing

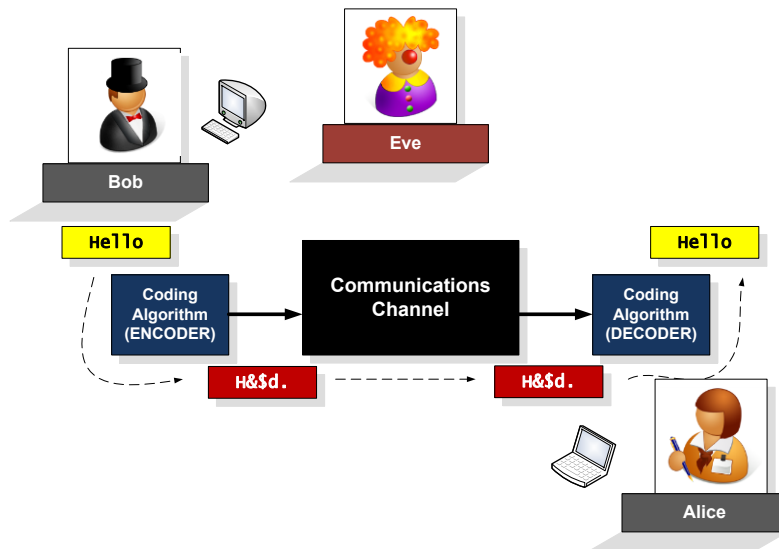


Figure 3.1 Bob, Alice and Eve

3.3 Simple cipher methods

One method of converting a message into cipher text is for Bob and Alice to agree on some sort of algorithm which Bob will use to scramble his message, and then Alice will do the opposite to unscramble the scrambled message. An example of this is the Caesar code, where it is agreed by Bob and Alice that the letters of the alphabet will be moved by a certain number of positions to the left or the right. It is named as the Caesar code as it was first documented by Julius Caesar who used a 3-letter shift.

In the example in Figure 3.2 the letters for the code have been moved forwards by two positions, thus a 'c' becomes an 'A', thus a coded message of 'RFC' is decoded as 'the'. There are several problems with this type of coding, though. The main one is that it is not very secure as there are only 25 unique codings, thus it would be easy for someone to find out the mapping. An improvement is to scramble up the mapping, such as in a code mapping (Figure 3.3), where a random mapping is used to deter the conversion. As there are more mappings, it improves the security of the code (4.03×10^{26} mappings), but it is still seen as being insecure as the probability of the letter in the mapped code is typically a pointer to the mapping. For the code in Figure 3.3, an 'A' appears most often, thus it is likely to be an 'e', which is the most probably letter in written English. Next 'Q' appears four times, thus this could be a 't', which is the next most probable. A more formal analysis of the probabilities is given in Table 3.1, where the letter 'e' is the most probable, followed by 't', and then 'o', and so on. It is also possible to look at two-letter occurrences (digrams), or at three-letter occurrences (trigrams), or even with words, where 'the' is the most common word.

A code mapping encryption scheme is easy to implement, but, unfortunately, once it has been 'cracked', it is easy to decrypt the encrypted data. Normally this type of cipher is implemented with an extra parameter which changes its mapping, such as changing the code mapping over time depending on the time-of-day and/or date. Thus parties which are allowed to decrypt the message know the mappings of the

code for a given time and/or date. For example, each day of the week could have a different code mapping.

📖 **Web link:** http://buchananweb.co.uk/flash_coding_shifted.html

📖 **Web link:** <http://buchananweb.co.uk/security20.aspx>

📖 **Web link:** <http://buchananweb.co.uk/security30.aspx>

Caesar code

abcdefghijklmnopqrstuvwxyz
YZABCDEF GHIJKLMNOPQRSTUVWXYZ

RFC ZMW QRM MB ML RFC ZSPLGLE B CAI

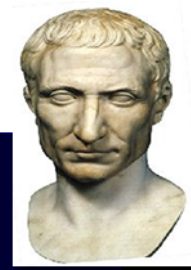


Figure 3.2 Caesar code

Code mapping

abcdefghijklmnopqrstuvwxyz
MGQOAFZBCDIEHXJKLNTQRWSUVY

QBCT CT MX AUMHKEA KCAQA JF QAUQ

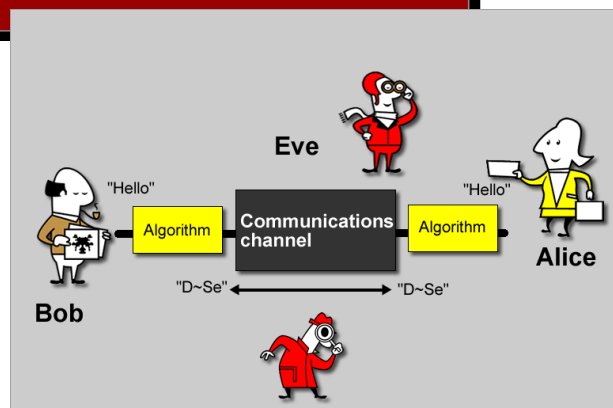


Figure 3.3 Code mapping

📖 **Web link:** <http://buchananweb.co.uk/security26.aspx>

Table 3.1 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

3.3.1 Vigenère cipher

An improved code was developed by Vigenère, where a different row is used for each character cipher, and is *polyalphabetic* cipher as it uses a number of cipher alphabets. Then the way that the user moves between the rows must be agreed before encryption. This can be achieved with a code word, which defines the sequence of the rows. For example the codeword **GREEN** could be used which defines that the rows used are: Row 6 (G), Row 17 (R), Row 4 (E), Row 4 (E), Row 13 (N), Row 6 (G), Row 17 (R), and so on (see Table 3.2). Thus the message is converted as:

Keyword **GREENGREENREE**
Plaintext **hellohowareyou**
Ciphertext **NVPPBNFAEEKPSY**

The great advantage of this type of code is that the same plaintext character will be coded with different values, depending on the position of the keyword. For example, for a keyword is GREEN, 'e' can be encrypted as 'K' (for G), 'V' (for R), 'I' (for E) and 'R' (for N). To improve security, the greater the size of the code word, the more the rows that can be included in the encryption process. Also, it is not possible to decipher the code by simple frequency analysis, as letters will change their coding depending on the current position of the keyword. 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).

Table 3.2 Coding

Plain	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

📖 Web link: http://buchananweb.co.uk/flash_vin.html

📖 Web link: <http://buchananweb.co.uk/security27.aspx>

📖 Web link: <http://buchananweb.co.uk/security29.aspx>

3.3.2 Homophonic substitution code

A homophonic substitution code overcomes the problems of frequency analysis of code, as it assigns a number of codes to a character which relates 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 3.3.

With this, each of the codes is assigned at random for each of the letters, with the number of codes assigned relating to the probability of their occurrence. Thus, using the code table in Table 3.3, the code mapping 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, each code will roughly have the same probability, thus it is not possible 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’ normally is represented by a single code, and there are three codes representing a ‘u’. Thus, if the ciphertext contains a code followed by one of three codes, then it is likely that the plaintext is a ‘q’ and a ‘u’.

A homophonic cipher is a monoalphabetic code, as it only uses one translation for the code mappings (even though several codes can be used for a single plaintext letter). This type of alphabet remains constant, whereas a polyalphabet can change its mapping depending on a variable keyword.

Table 3.3 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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

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 is that, like the bit rotate operators, it preserves the information in the processed output, and can be undone merely by operating on the output with the value that was used to process the value. 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

which results in the original value. Thus 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.

The other operator is **mod**, which returns the remainder of a division operation. For example 29 mod 7 gives 1.

3.5 Key-based cryptography

The main objective of cryptography is to provide a mechanism for two (or more) entities to communicate without any other entity being able to read or change the message. Along with this it can provide other services, such as:

- **Integrity check.** This makes sure that the message has not been tampered with by non-legitimate sources.
- **Providing authentication.** This verifies the sender identity. Unfortunately most of the current Internet infrastructure has been build on a fairly open system, where users and devices can be easily spoofed, thus authentication is now a major factor in verifying users and devices.

One of the main problems with using a secret algorithm for encryption is that it is difficult to determine if Eve has found-out the algorithm used, thus most encryption methods use a key-based approach where an electronic key is applied to a well-known algorithm. Another problem with using different algorithms for the encryp-

tion is that it is often difficult to keep devising new algorithms and also to tell the receiving party that the text is being encrypted with the new algorithm. Thus, using electronic keys, there are no problems with everyone having the encryption/decryption algorithm, because without the key it should be computationally difficult to decrypt the message (Figure 3.4).

The three main methods of encryption are (Figure 3.5):

- **Symmetric key-based encryption.** This involves the same key being applied to the encrypted data, in order that the original data is recovered. Typical methods are DES, 3DES, RC2, RC4, AES, and so on.
- **Asymmetric key-based encryption.** This involves using a different key to decrypt the encrypted data, in order that the original data is recovered. A typical method is RSA, DSA and El Gamal.
- **One-way hash functions.** With this it is not possible to recover the original source information, but the mapping between the value and the hashed value is known. The one-way hash function is typically used in authentication applications, such as generating a hash value for a message, and will be covered in Unit 4. The two main methods are MD5 and SHA-1, and it is also used in password hashing applications, where a password is hashed with a one-way function, and the result is stored. This is the case in Windows and UNIX login, where the password is stored as a hash value. Unfortunately, if the password is not a strong one, the hash value is often prone to a dictionary-type attack, where an intruder tries many different passwords and hashes them, and then compares it with the stored one.

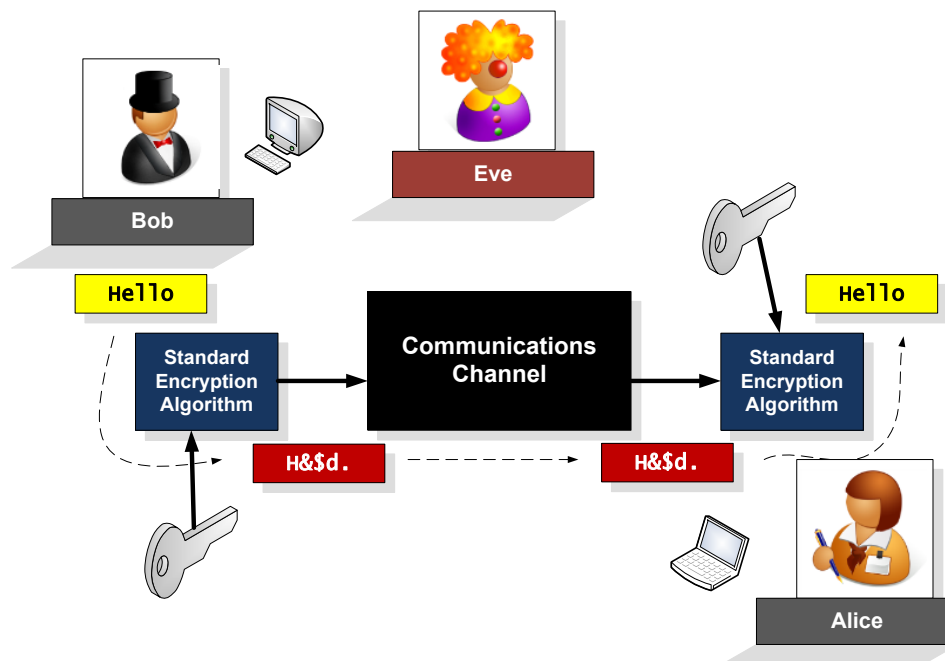


Figure 3.4 Key-based encryption

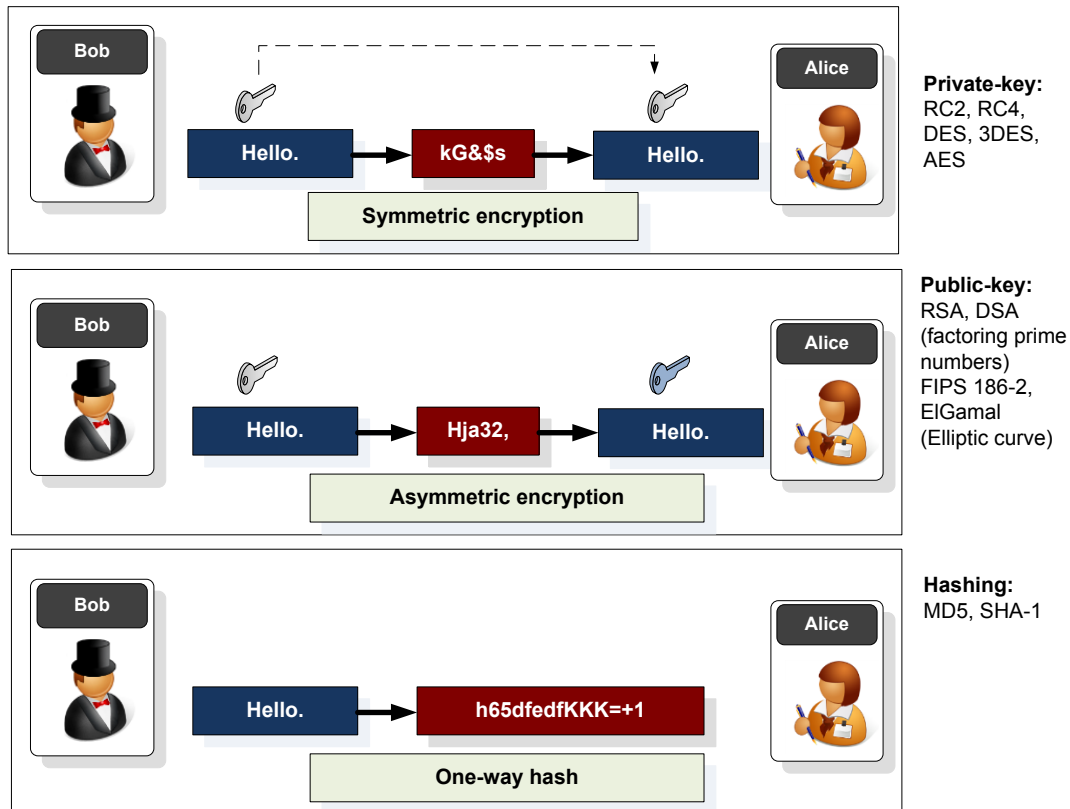


Figure 3.5 Encryption methods

3.5.1 Computation difficulty

Every code is crackable and the measure of the security of a code is the amount of time it takes a person not addressed in the code to break it. Unless there are weaknesses in the encryption algorithm, the normal way to break cipher text is where a computer tries all the possible keys, until it finds a match. Thus a 1-bit code would only have two keys; a 2-bit code would have four keys; and so on. Table 3.4 shows the number of possible keys, as a function of the number of bits in the key. For example it can be seen that a 64-bit code has 184000000000000000 different keys. Thus if one key is tested every 10 μ s then it would take 1.84×10^{14} seconds (5.11×10^{10} hours or 2.13×10^8 days or 5834602 years). So, for example, if it takes 1 million years for a person to crack the code, it can be considered safe. Unfortunately, from the point of security of an encrypted message, the performance of computer systems increases by the year. For example, if a computer takes 1 million years to crack a code, then assuming an increase in computing power of a factor of two per year, it would take 500000 years the next year. Then, Table 3.3 shows that after almost 20 years it would take only 1 year to decrypt the same message. This is a worrying factor as encryption algorithms which are used in the financial applications, which was one of the first after the military to adopt encryption, are now over 30 years old².

The increasing power of computers is one factor in reducing the processing time; another is the increasing usage of parallel processing, as data decryption is well suit-

² DES is the standard encryption algorithm used in financial transactions and was first published in 1977.

ed to parallel processing as each processor element can be assigned a number of keys to check the encrypted message. Each of them can then work independently of the other³. Table 3.6 gives typical times, assuming a doubling of processing power each year, for processor arrays of 1, 2, 4...4096 elements. It can thus be seen that with an array of 4096 processing elements it takes only seven years before the code is decrypted within two years. Thus an organization which is serious about deciphering messages is likely to have the resources to invest in large arrays of processors, or networked computers. It is also likely that many governments have computer systems which have thousands of processors, operating in parallel.

Table 3.4 Number of keys related to the number of bits in the key

<i>Code size</i>	<i>Number of keys</i>	<i>Code size</i>	<i>Number of keys</i>	<i>Code size</i>	<i>Number of keys</i>
1	2	12	4 096	52	4.5×10^{15}
2	4	16	65 536	56	7.21×10^{16}
3	8	20	1 048 576	60	1.15×10^{18}
4	16	24	16 777 216	64	1.84×10^{19}
5	32	28	2.68×10^8	68	2.95×10^{20}
6	64	32	4.29×10^9	72	4.72×10^{21}
7	128	36	6.87×10^{10}	76	7.56×10^{22}
8	256	40	1.1×10^{12}	80	1.21×10^{24}
9	512	44	1.76×10^{13}	84	1.93×10^{25}
10	1 024	48	2.81×10^{14}	88	3.09×10^{26}

Table 3.5 Time to decrypt a message assuming an increase in computing power

<i>Year</i>	<i>Time to decrypt (years)</i>	<i>Year</i>	<i>Time to decrypt (years)</i>
0	1 million	10	977
1	500 000	11	489
2	250 000	12	245
3	125 000	13	123
4	62 500	14	62
5	31 250	15	31
6	15 625	16	16
7	7 813	17	8
8	3 907	18	4
9	1 954	19	2

Table 3.6 Time to decrypt a message with increasing power and parallel processing

<i>Processors</i>	<i>Year 0</i>	<i>Year 1</i>	<i>Year 2</i>	<i>Year 3</i>	<i>Year 4</i>	<i>Year 5</i>	<i>Year 6</i>	<i>Year 7</i>
1	1 000 000	500 000	250 000	125 000	62 500	31 250	15 625	7 813
2	500 000	250 000	125 000	62 500	31 250	15 625	7 813	3 907
4	250 000	125 000	62 500	31 250	15 625	7 813	3 907	1 954
8	125 000	62 500	31 250	15 625	7 813	3 907	1 954	977
16	62 500	31 250	15 625	7 813	3 907	1 954	977	489
32	31 250	15 625	7 813	3 907	1 954	977	489	245
64	15 625	7 813	3 907	1 954	977	489	245	123
128	7 813	3 907	1 954	977	489	245	123	62
256	3 906	1 953	977	489	245	123	62	31
512	1 953	977	489	245	123	62	31	16
1 024	977	489	245	123	62	31	16	8

³ This differs from many applications in parallel processing which suffer from interprocess(or) communication

2048	488	244	122	61	31	16	8	4
4096	244	122	61	31	16	8	4	2

3.5.2 Cracking the code

A cryptosystem normally converts plaintext into ciphertext, using a key. There are several methods that an intruder can use to crack a code, including:

- **Exhaustive search.** Where the intruder uses brute force to decrypt the ciphertext and tries every possible key (Figure 3.6).
- **Known plaintext attack.** Where the intruder knows part of the ciphertext and the corresponding plaintext. The known ciphertext and plaintext can then be used to decrypt the rest of the ciphertext (Figure 3.7).
- **Man-in-the-middle.** Where the intruder is hidden between two parties and impersonates each of them to the other (Figure 3.8).
- **Chosen-ciphertext.** Where the intruder sends a message to the target, this is then encrypted with the target's private-key and the intruder then analyses the encrypted message. For example, an intruder may send an e-mail to the encryption file server and the intruder spies on the delivered message.
- **Active attack.** Where the intruder inserts or modifies messages (Figure 3.9).
- **The replay system.** Where the intruder takes a legitimate message and sends it into the network at some future time (Figure 3.10).
- **Cut-and-paste.** Where the intruder mixes parts of two different encrypted messages and, sometimes, is able to create a new message. This message is likely to make no sense, but may trick the receiver into doing something that helps the intruder.
- **Time resetting.** Some encryption schemes use the time of the computer to create the key. Resetting this time or determining the time that the message was created can give some useful information to the intruder.
- **Time attack.** This involves determining the amount of time that a user takes to decrypt the message; from this the key can be found.

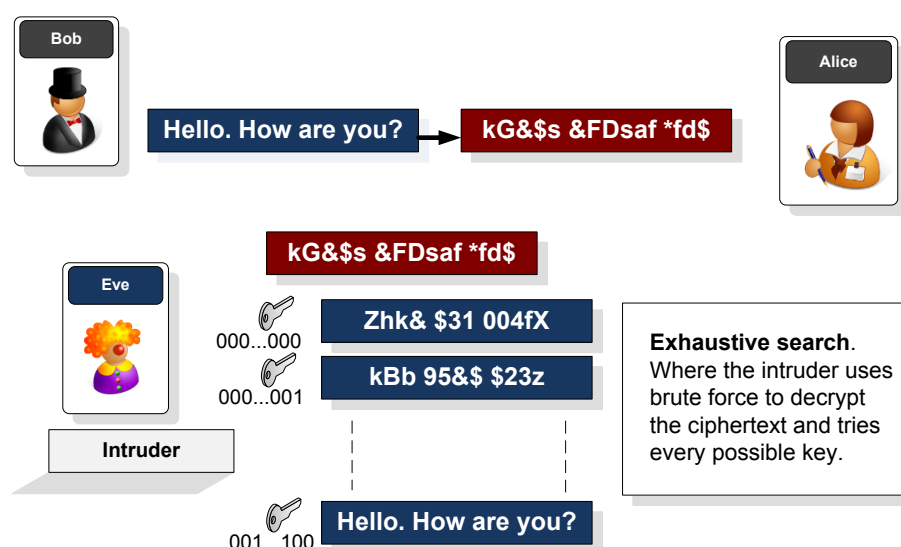


Figure 3.6 Exhaustive search

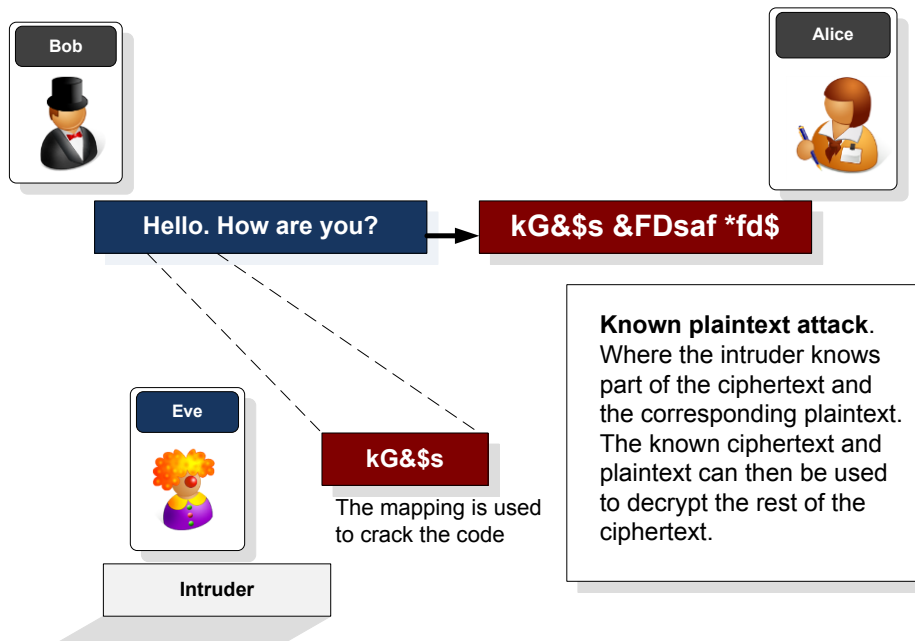


Figure 3.7 Known plaintext attack

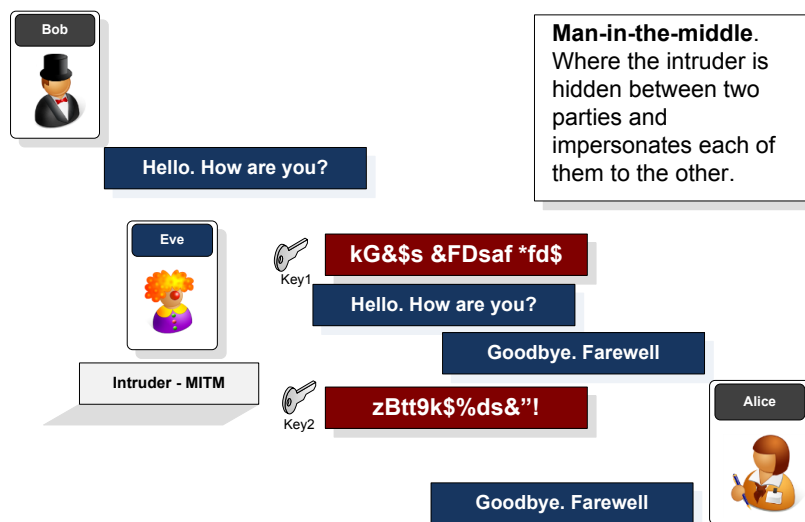


Figure 3.8 Man-in-the-middle

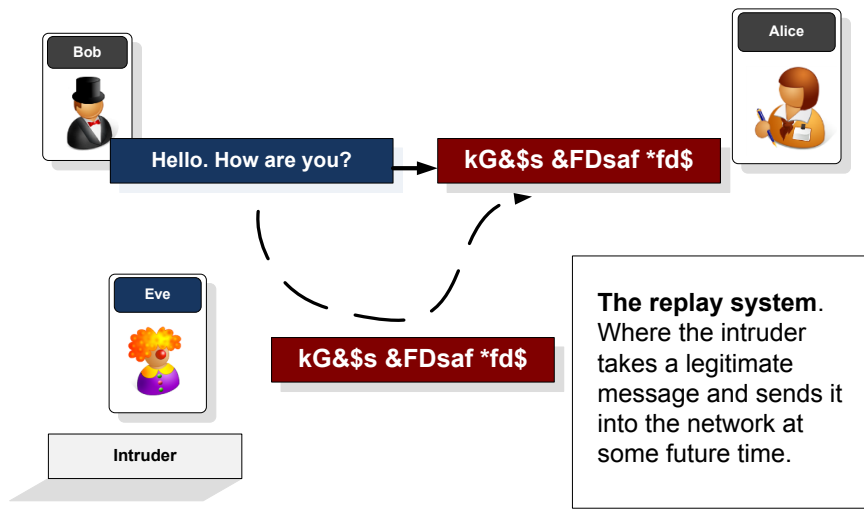


Figure 3.9 Replay attack

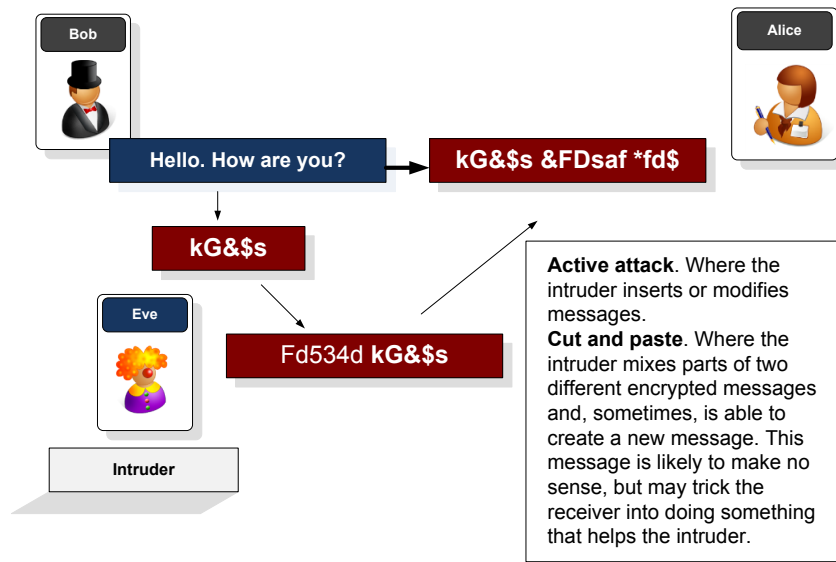





Figure 3.10 Active attack

3.5.3 Stream encryption and block encryption

The encryption method can either be applied by selecting blocks of a data, and then encrypting them, or it can operate on a data stream, where one bit at a time is encrypted (Figure 3.11). Typical block sizes are 128, 192 or 256 bits. Overall stream encryption is much faster, and can typically be applied in real-time applications. For example, stream-based encryption is used with wireless systems, where an infinite key is created from the wireless key. This is then exclusive-OR-ed with the data stream, to produce the ciperstream. The main methods are (Figure 3.11 and Figure 3.12):

- **Stream encryption:** RC4 (one of the fastest streaming algorithms around).
- **Block encryption:** RC2 (40-bit key size), RC5 (variable block size), IDEA, DES, 3DES, AES (Rijndael), Blowfish and Twofish.

3DES: **Web link:** <http://buchananweb.co.uk/security07.aspx>

RC2:  **Web link:** <http://buchananweb.co.uk/security06.aspx>
 AES:  **Web link:** <http://buchananweb.co.uk/security15.aspx>
 RSA:  **Web link:** <http://buchananweb.co.uk/security08.aspx>

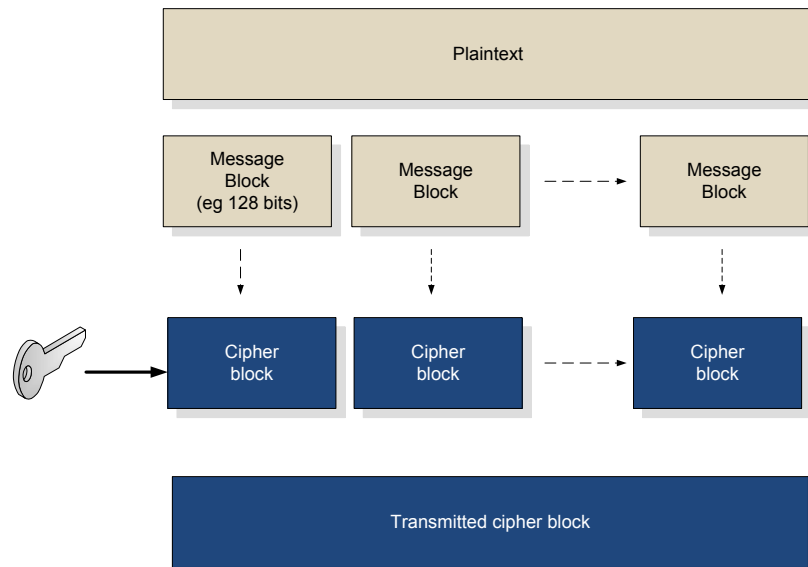


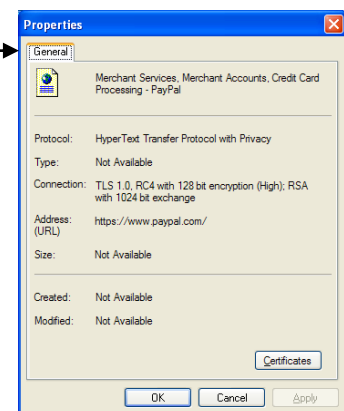
Figure 3.11 Block coding

The most widely used private-key encryption (symmetric) algorithms are:

- RC2 (40-bit key size, 64-bit blocks).
- RC4 (stream cipher) – used in SSL and WEP.
- RC5 (variable key size, 32, 64 or 128 bit block sizes).
- AES (128, 192 or 256 bit key size, 128 bit block size).
- DES (56 bit key size, 64 bit block size).
- 3DES (168 bit key size, 64 bit block size).

An example of a stream conversion is:

Data stream: 0101110101010111
 Pseduo-infinite key: 1001100000111010
 Result: **1100010101101101**



where the receive will then generate the same infinite key, and simply X-OR it with the received stream to recover the data stream. A weakness of the system is obviously in the way that the pseduo-infinite key, which is typically generated from a pass phrase (which limits the actual range of keys). To overcome the same pseduo-infinite key being used for different communications, an initialization vector (IV) is normally used (the random seed). This can then be incremented each for each data frame sent, and will thus result in a different key for each transmission. Unfortunately the IV value has a limited range, and will enventually roll-over to the same value, after-which an intruder can use a statisical analysis technique to crack the code.

See: http://ceres.napier.ac.uk/staff/bill/wireless_security/wireless_security.htm

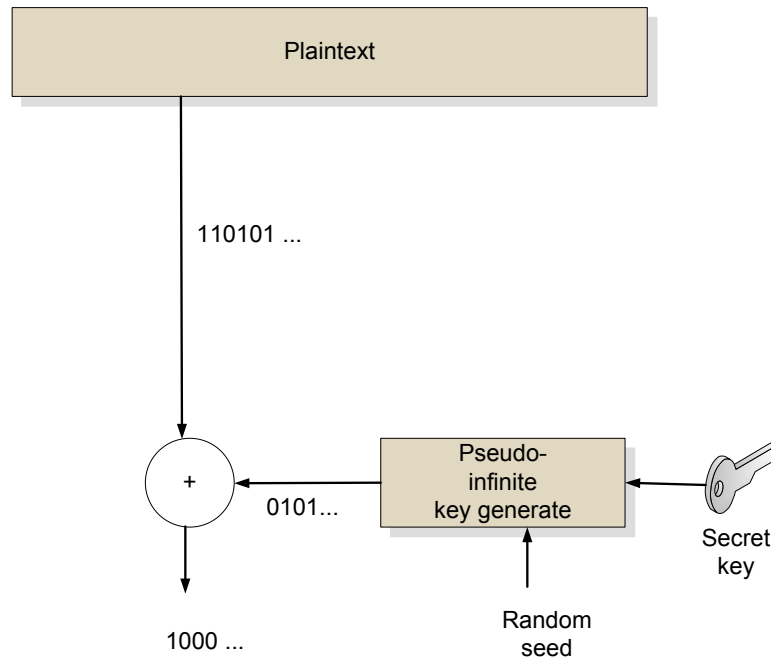


Figure 3.12 Stream coding

3.6 Brute-force analysis

It is important to understand how well cipher text will cope with a brute force attack, where an intruder tries all the possible keys. As an example, let's try a 64-bit encryption key which gives us: 1.84×10^{19} combinations (2^{64}). If we now assume that we have a fast processor that tries one key every billionth of second (1GHz clock), then the average⁴ time to crack the code will be:

$$T_{average} = 1.84 \times 10^{19} \times 1 \times 10^{-9} \div 2 \approx 9,000,000,000 \text{ seconds}^5$$

It will thus take approximately 2.5 million hours (150 million minutes or 285 years) to crack the code, which is likely to be strong enough in most cases. Unfortunately as we have seen, the computing power often increases by the year, so if we assume a doubling of computing power, then:

Date	Hours	Days	Years
0	2,500,000	104,167	285
+1	1,250,000	52,083	143
+2	625,000	26,042	71
+3	312,500	13,021	36
+4	156,250	6,510	18
+5	78,125	3,255	9
+6	39,063	1,628	4
+7	19,532	814	2

⁴ The average time will be half of the maximum time

⁵ 9,223,372,036 seconds to be more precise

+8	9,766	407	1
+9	4,883	203	1
+10	2,442	102	0.3
+11	1,221	51	0.1
+12	611	25	0.1
+13	306	13	0
+14	153	6	0
+15	77	3	0
+16	39	2	0
+17	20	1	0

we can see that it now only takes 17 years to crack the code in a **single day**! If we then apply parallel processing, the time to crack reduces again. In the following an array of 2×2 (4 processing elements), 4×4 (16 processing elements), and so on, are used to determine the average time taken to crack the code. If, thus, it currently takes 2,500,000 minutes to crack the code, it can be seen that by Year 6, it takes less than one minute to crack the code, with a 256×256 processing matrix.

Processing Elements	Year 0 (minutes)	Year 1 (min)	Year 2 (min)	Year 3 (min)	Year 4 (min)	Year 5 (min)	Year 6 (min)	Year 7 (min)
1	2500000	1250000	625000	312500	156250	78125	39062.5	19531.3
4	625000	312500	156250	78125	39062.5	19531.3	9765.7	4882.9
16	156250	78125	39062.5	19531.3	9765.7	4882.9	2441.5	1220.8
64	39063	19531.5	9765.8	4882.9	2441.5	1220.8	610.4	305.2
256	9766	4883	2441.5	1220.8	610.4	305.2	152.6	76.3
1024	2441	1220.5	610.3	305.2	152.6	76.3	38.2	19.1
4096	610	305	152.5	76.3	38.2	19.1	9.6	4.8
16384	153	76.5	38.3	19.2	9.6	4.8	2.4	1.2
65536	38	19	9.5	4.8	2.4	1.2	0.6	0.3

The use of parallel processing is now well-known in the industry, and the Electronic Frontier Foundation (EFF) set out to prove that DES was weak, and created a 56-bit DES crack which had an array of 29 circuits of 64 chips (1856 elements), and processed 90,000,000 keys per seconds. It, in 1998, eventually cracked the code within 2.5 days. A more recent machine is the COPACOBANA (Cost-Optimized Parallel CODE Breaker) which costs less than \$10,000, and can crack a 64-bit DES code in less than nine days.

The ultimate in distributed applications is to use unused processor cycles of machines connected to the Internet. For this applications such as **distributed.net** allow the analysis of a key space when the screen saver is on (Figure 3.13). It has since used the method to crack a number of challenges, such as in 1997 with a 56-bit RC5 Encryption Challenge. It was cracked in 250 days, and has since moved on, in 2002, to crack 64-bit RC5 Encryption Challenge in 1,757 days (with 83% of the key space tested). The current challenge involves a 72-bit key.

Along with increasing power of computers, and parallel processing, another method of improving the performance of brute force analysis is to use supercomputers. Two of the most powerful machines in the world are:

- **BlueGene/L – eServer Blue Gene Solution.** DOE/NNSA/LLNL, IBM Department of Energy's (DOE) National Nuclear Security Administration's (NNSA) which has 131,072 processors, and gives a throughput of 367,000 Gigaflop= 367,000,000 Mflops (which is 1,835,000 times more powerful than a desktop). The University of Edinburgh has just deployed their new BlueGene and runs at 60,000 Gigaflops.
- **Red Storm - Sandia/ Cray Red Storm.** NNSA/Sandia National Laboratory United States. It has a 2.4 GHz dual core from Cray Inc and has 26,544 processors with an operating throughput of 127,000 Gflops.

An encryption algorithm which is cracked in a million minutes on a standard PC, could BlueGene less than a minute to crack.

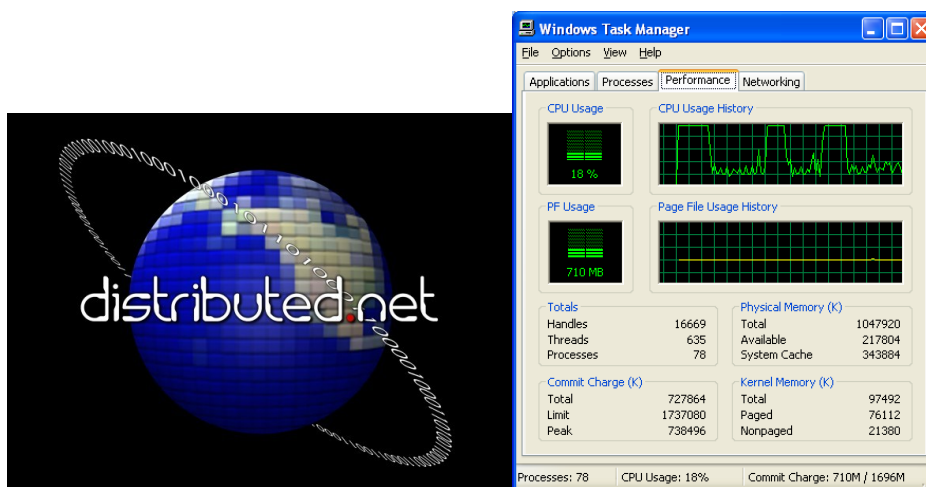


Figure 3.13 Distributed.net

3.7 Public-key, private-key and session keys

The encryption process can either use a public key or a secret key (Figure 3.4). With a secret key, the key is only known to the two communicating parties (symmetric key-based encryption). This key can be fixed or can be passed from the two parties over a secure communications link (perhaps over the postal network or a leased line). The two popular private key techniques are **DES** (Data Encryption Standard) and **IDEA** (International Data Encryption Algorithm).

In public-key encryption, each entity has both a public and a private key (asymmetric key-based encryption). The two entities then communicate using each other's public keys. Normally, in a public-key system, each user uses a public enciphering transformation which is widely known and a private deciphering transform which is known only to that user. The private transformation is described by a private key, and the public transformation by a public key derived from the private key by a one-way transformation. The RSA (after its inventors Rivest, Shamir and Adleman) technique is one of the most popular public-key techniques and is based on the difficulty of factoring large numbers.

Another important factor is the time relevance of the generated keys (whether symmetric or asymmetric keys), where the keys could be fixed for a range of connec-

tions, and have some form of key regeneration after a certain number of connections, or for a certain time limit. They can also be sessional, where the keys are defined for each session. The advantage with sessional keys is that they typically do not have to be as long as non-time based keys, as the session typically only occurs for a short time, after which new keys are regenerated. Thus with brute force the intruder might only be able to get the details of a single session, by which time it is probably too late to gain and useful information from it. In wireless communications, the WEP encryption standard uses a fixed key, based on a pass phrase, and is used by all the nodes on the network. Thus, once the key has been cracked it can be used to decrypt all the communications for the network. An improvement on this is to use TKIP (which is part of WPA), which uses a session key for each connection, and it is thus much more difficult to crack. Both these techniques use the RC4 encryption method, which uses stream encryption. Newer systems are likely to be based around WPA-2 which uses a block encryption standard (AES).

3.8 Adding salt

A major problem in encryption is playback where an intruder can copy an encrypted message and play it back, as the same plain text will always give the same cipher text. The solution is to add **salt** to the encryption key, as that it changes its operation from block-to-block (for block encryption) or data frame-to-data frame (for stream encryption). The Electronic Code Book (ECB) method is weak, as the same cipher text appears for the same blocks. For example:

```
Hello -> 5ghd%43f=  
Hello -> 5ghd%43f=
```

If the intruder knew that the plaintext was “Hello”, they would be able to play back this message. This solution to this is to add salt. This is typically done with an IV (Initialisation Vector) which must be the same on both sides. In WEP, the IV is incremented for each data frame, so that the cipher text changes. As can be seen in Figure 3.15, in blocks of the same data will be encrypted in the same way. An improvement is to use Cipher Block Chaining (CBC). This method uses the IV for the first block, and then the results from the previous block to encrypt the current block.

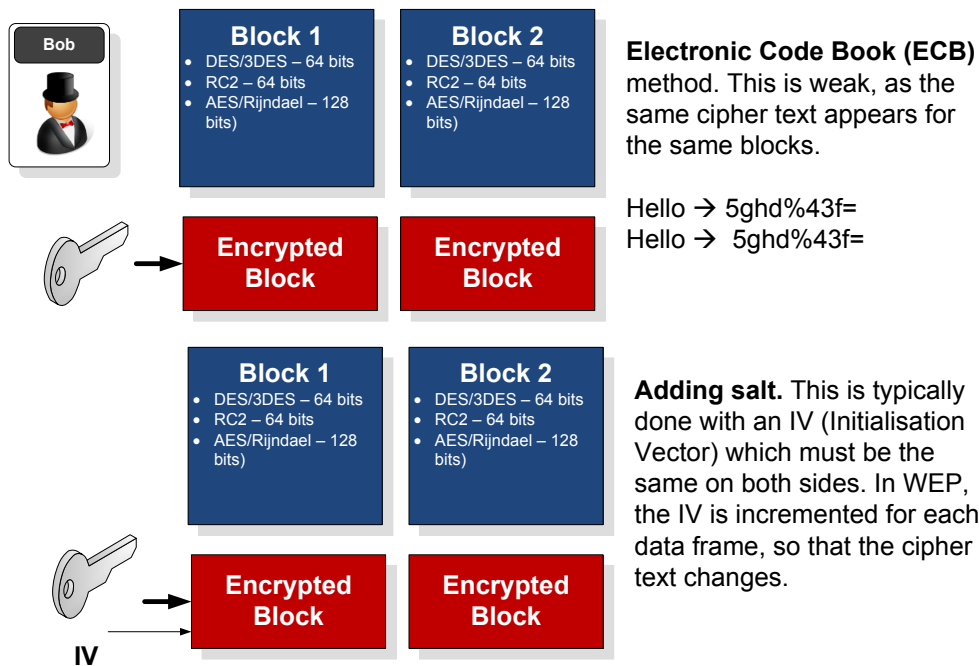


Figure 3.14 ECB and adding salt

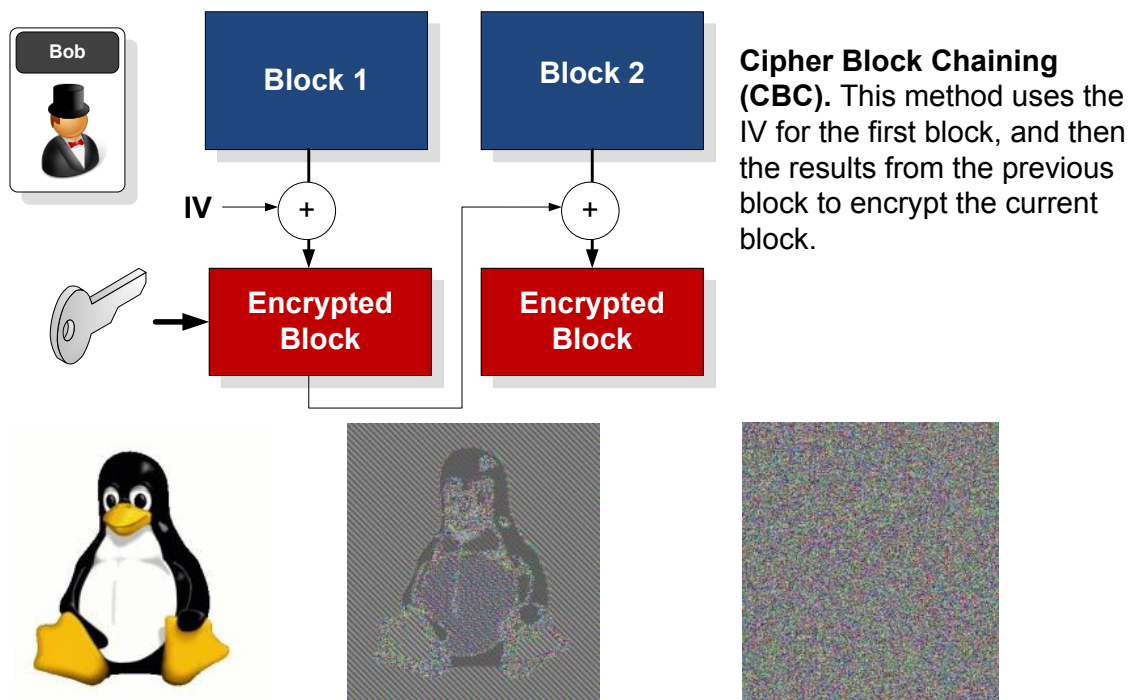


Figure 3.15 ECB and CBC

3.9 Private-key encryption

Private-key (or secret-key) encryption techniques use a secret key which is only known by the two communicating parties, as illustrated in Figure 3.16. This key can be generated by a phase-phase, or can be passed from the two parties over a secure communications link. The most popular private-key techniques include:

- **DES.** DES (Data Encryption Standard) is a block cipher scheme which operates on 64-bit block sizes. The private key has only **56 useful bits**, as eight of its bits are used for parity (which gives 2^{56} or 10^{17} possible keys). DES uses a complex series of permutations and substitutions, the result of these operations is XOR'ed with the input. This is then repeated 16 times using a different order of the key bits each time. DES is a strong code and has never been broken, although several high-powered computers are now available which, using brute force, can crack the code. A possible solution is **3DES** (or triple DES) which uses DES three times in a row. First to encrypt, next to decrypt and finally to encrypt. This system allows a key-length of more than 128 bits. The technique uses two keys and three executions of the DES algorithm. A key, K_1 , is used in the first execution, then K_2 is used and finally K_1 is used again. These two keys give an effective key length of 112 bits, that is 2×64 key bits minus 16 parity bits. The Triple DES process is illustrated in Figure 3.17.
- **RC4.** RC4 is a **stream** cipher designed by RSA Data Security, Inc and was a secret until information on it appeared on the Internet. The secure socket layer (SSL) protocol and wireless communications (IEEE 802.11a/b/g) use RC4. It uses a pseudo random number generator, where the output of the generator is XOR'ed with the plaintext. It is a fast algorithm and can use any key-length. Unfortunately the same key cannot be used twice. Recently a 40-bit key version was broken in eight days without special computer power.
- **AES/Rijndael.** AES (Advanced Encryption Standard) is a new standard for encryption, and uses 128, 192 or 256 bits. It was selected by NIST in 2001 (after a five year standardisation process). The name Rijndael comes from its Belgium creators: Joan Daemen and Vincent Rijmen. The future of wireless systems (WPA-2) is likely to be based around AES (while WPA uses TKIP which is a session key method which is based around stream encryption using RC4).
- **IDEA.** IDEA (International Data Encryption Algorithm) is similar to DES. It operates on 64-bit blocks of plaintext, using a 128-bit key, and has over 17 rounds with a complicated mangler function. During decryption this function does not have to be reversed and can simply be applied in the same way as during encryption (this also occurs with DES). IDEA uses a different key expansion for encryption and decryption, but every other part of the process is identical. The same keys are used in DES decryption, but in the reverse order. The key is devised in eight 16-bit blocks; the first six are used in the first round of encryption the last two are used in the second run. It is free for use in non-commercial version and appears to be a strong cipher.
- **RC5.** RC5 is a fast block cipher designed by Rivest for RSA Data Security. It has a parameterized algorithm with a variable block size (32, 64 or 128 bits), a variable key size (0 to 2048 bits) and a variable number of rounds (0 to 255). It has a heavy use of data dependent rotations, and the mixture of different operations, which assures that RC5 is secure.

The major advantage that private-key encryption has over public-key is that it is typically much faster to decrypt, and can thus be used where a fast conversion is required, such as in real-time encryption.

- 📖 Web link: <http://buchananweb.co.uk/security07.aspx> [3DES]
 📖 Web link: <http://buchananweb.co.uk/security06.aspx> [RC2]
 📖 Web link: <http://buchananweb.co.uk/security15.aspx> [AES/Rijndael]

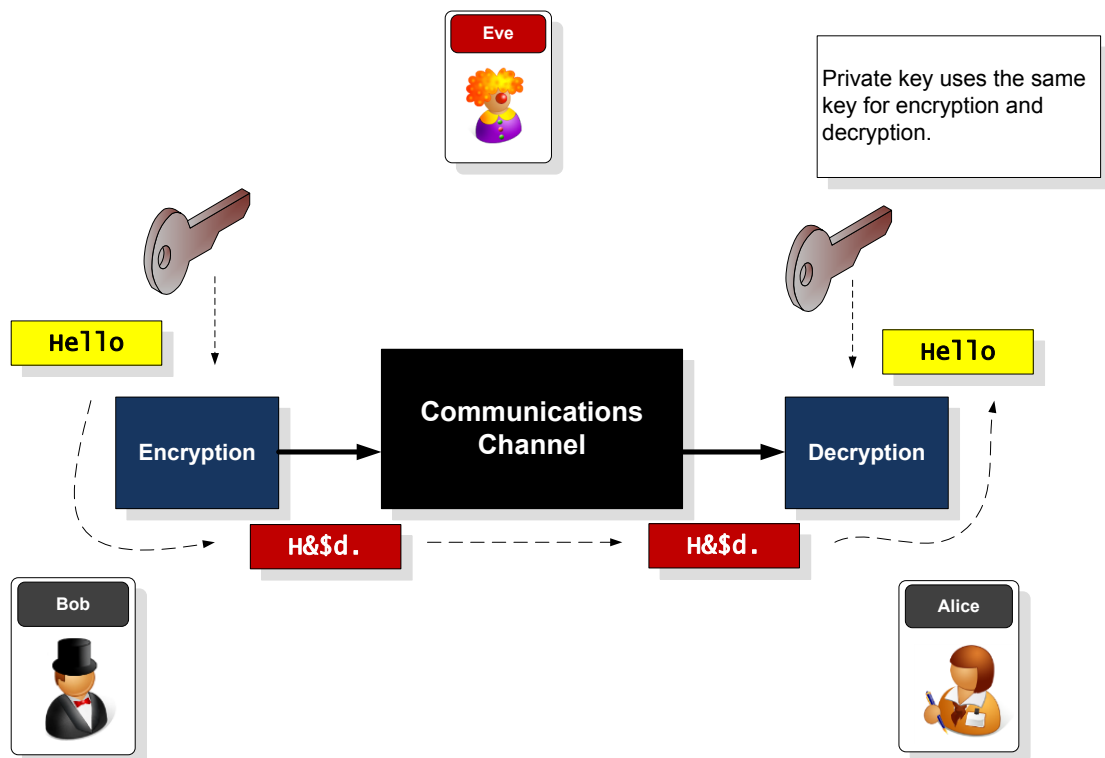


Figure 3.16 Private key encryption/decryption process

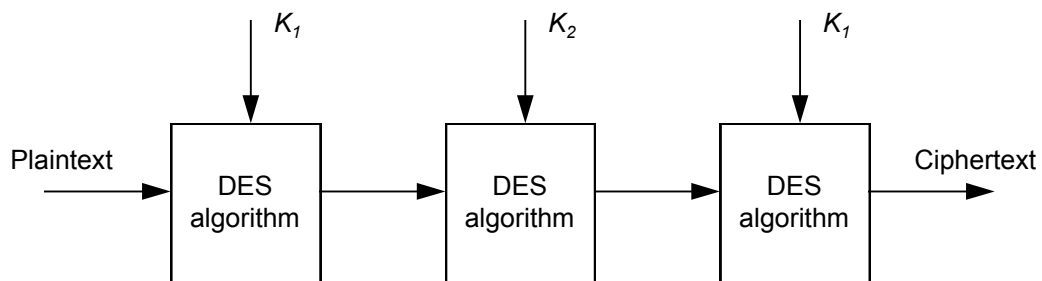


Figure 3.17 Triple DES process

3.10 Encryption classes

The .NET environment provides a number of cryptography classes. A good method is to use a code wrapper, which provides a simple method of accessing these classes [1]. It provides encryption algorithms such as DES, 3DES and BlowFish, and also hash algorithms such as MD5 and SHA (which will be covered in Chapter 4). The following is a simple example using the 3DES algorithm:

```
using System;
using XCrypt;
// Program uses XCrypt library from http://www.codeproject.com/csharp/xcrypt.asp
namespace encryption
{
    class MyEncryption
```

```

{
    static void Main(string[] args)
    {
        XCryptEngine xe = new XCryptEngine();
        xe.InitializeEngine(XCryptEngine.AlgorithmType.TripleDES);
        // other algorithms are:
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.BlowFish);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.Twofish);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.DES);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.MD5);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.RC2);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.Rijndael);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.SHA);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.SHA256);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.SHA384);
        // xe.InitializeEngine(XCryptEngine.AlgorithmType.SHA512);

        xe.Key = "MyKey"; // Define the public key
        Console.WriteLine("Enter string to encrypt:");
        string inText = Console.ReadLine();
        string encText = xe.Encrypt(inText);
        string decText = xe.Decrypt(encText);
        Console.WriteLine("Input: {0}\r\nEncr: {1}\r\nDecr: {2}",
                           inText, encText, decText);
        Console.ReadLine();
    }
}

```

A text string is used to define the key as it is easier to remember over a binary or hexadecimal define key.

A sample run with 3DES gives:

```

Enter string to encrypt:
test
Input: test
Encr: uVZLHJ3Wr8s=
Decr: test

```

By changing the method to SHA-1 (SHA) gives:

```

Enter string to hash: test
Input: test
Hash: qUqP5cyxm6YcTAhz05Hph5gvu9M=

```

The code for this simple example is available at:

<http://buchananweb.co.uk/encryption.zip>

3.10.1 Key interchange

The major problem of private-key encryption is how to pass the key between Bob and Alice, without Eve listening (Figure 3.18). This problem was solved by Whitfield Diffie in 1975, who created the Diffie-Hellman method. With this method, Bob and Alice generate two random values, and perform some calculations (Figure 3.16 and Figure 3.19), and pass the result of the calculations to each other (Figure 3.20). Once these values have been received at either end, Bob and Alice will have the same secret key, which Eve cannot compute (without extensive computation). Diffie-Hellman is used in many applications, such as in VPNs (Virtual Private Networks), SSH, and secure FTP. The following shows a trace of a connection to a secure FTP site:

```

STATUS:> Initializing SFTP21 module...
STATUS:> Resolving host name mysite.com ...

```

```

STATUS:> Host name mysite.com resolved: ip = 1.2.3.4.
STATUS:> Connecting to SFTP server ftp1.napier.ac.uk:22 (ip = 1.2.3.4 )
          Key Method: Diffie-Hellman-group1-SHA1
          Host Key Algorithm: SSH-RSA
          Session Cipher: 192 bit TripleDES-cbc
          Session MAC: HMAC-MD5
          Session Compressor/Decompressor: ZLIB
STATUS:> Getting working directory...
STATUS:> Home directory: /home/test

```

Where it can be seen that in this secure FTP transaction, the **encryption** being used is **3DES** (TripleDES), the message **authentication** method is **HMAC-MD5** (see Section 4.7) and the **key exchange** is Diffie-Hellman. Overall Diffie-Hellman has three groups: Group 1, Group 3 or Group 5. These determine the size of the prime number bases which are used in the key exchange, where Group 5 is more secure than Group 2, which is more secure than Group 1.

📖 **Web link:** http://buchananweb.co.uk/flash_diffie.html

📖 **Web link:** <http://buchananweb.co.uk/security02.aspx> [Diffie-Hellman example]

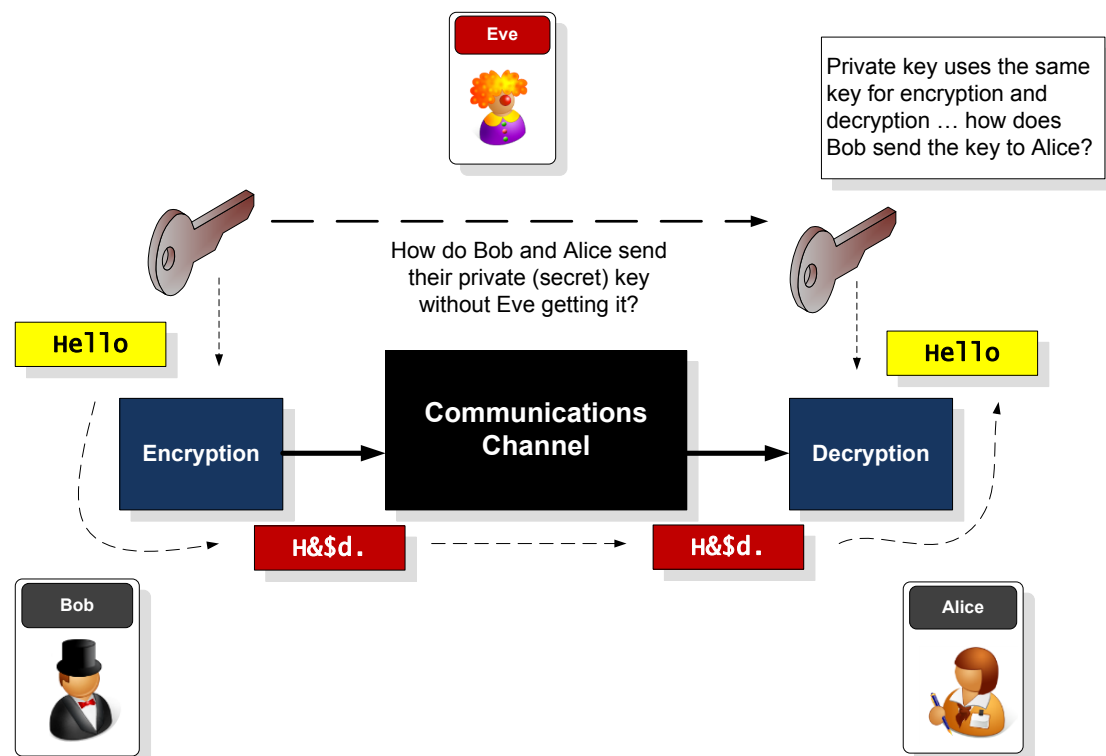


Figure 3.18 Private key encryption

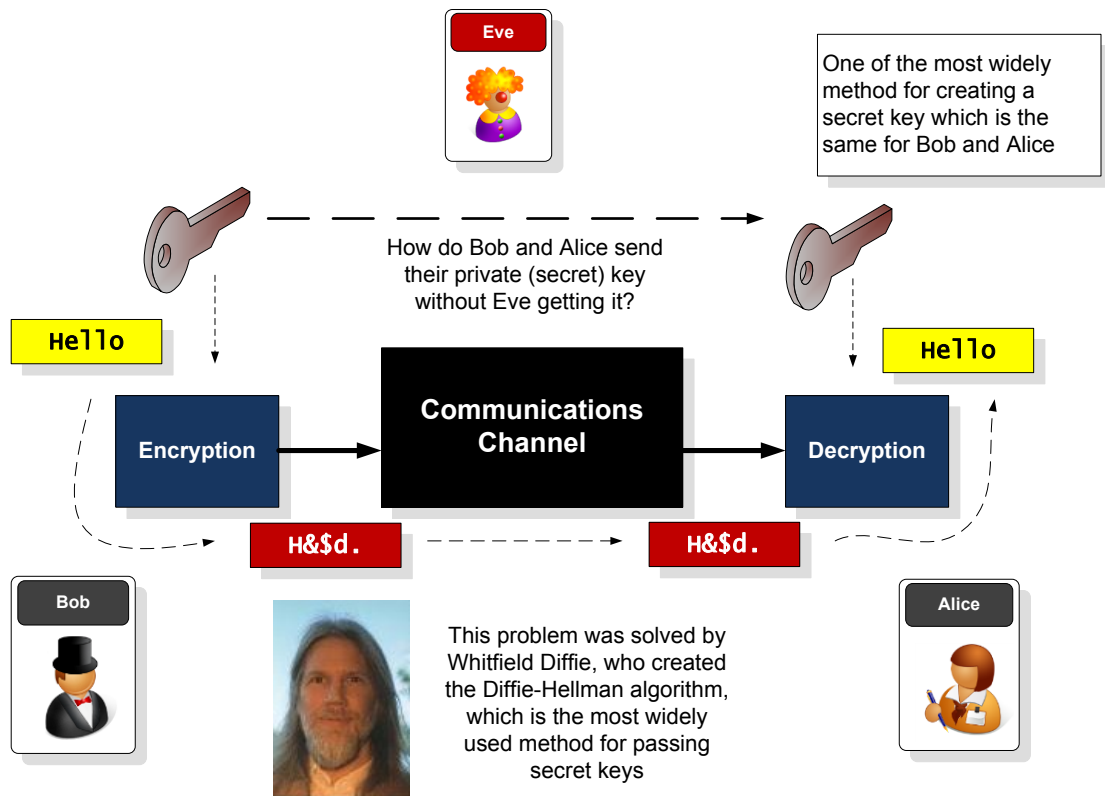


Figure 3.19 Diffie-Helman method

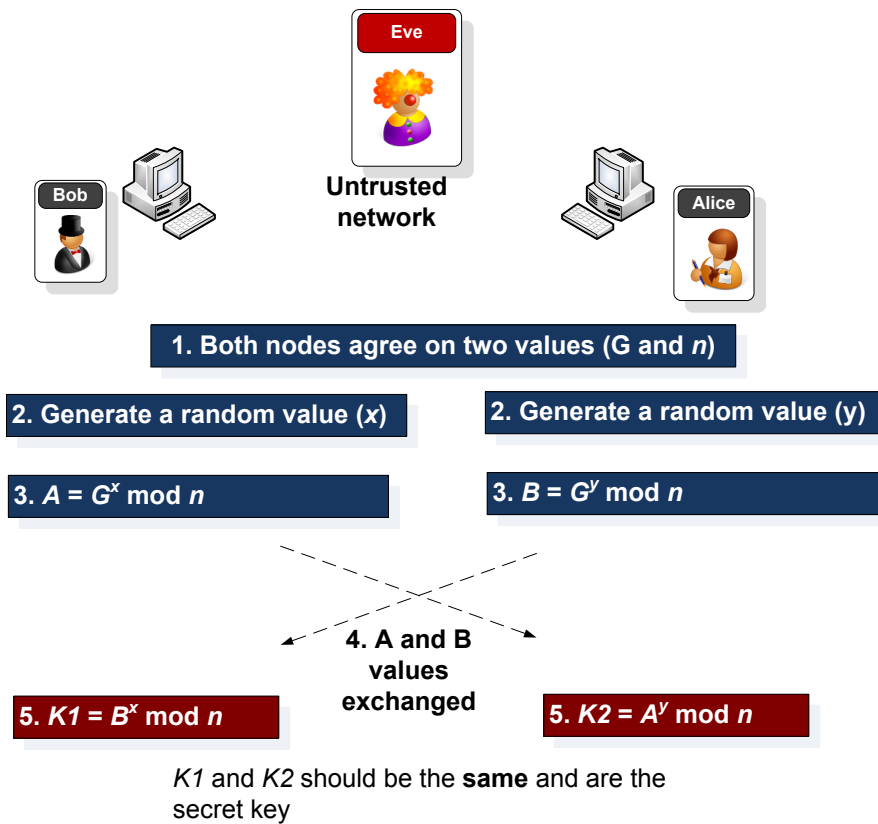


Figure 3.20 Diffie-Hellman process (see <http://buchananweb.co.uk/diffie.aspx>)

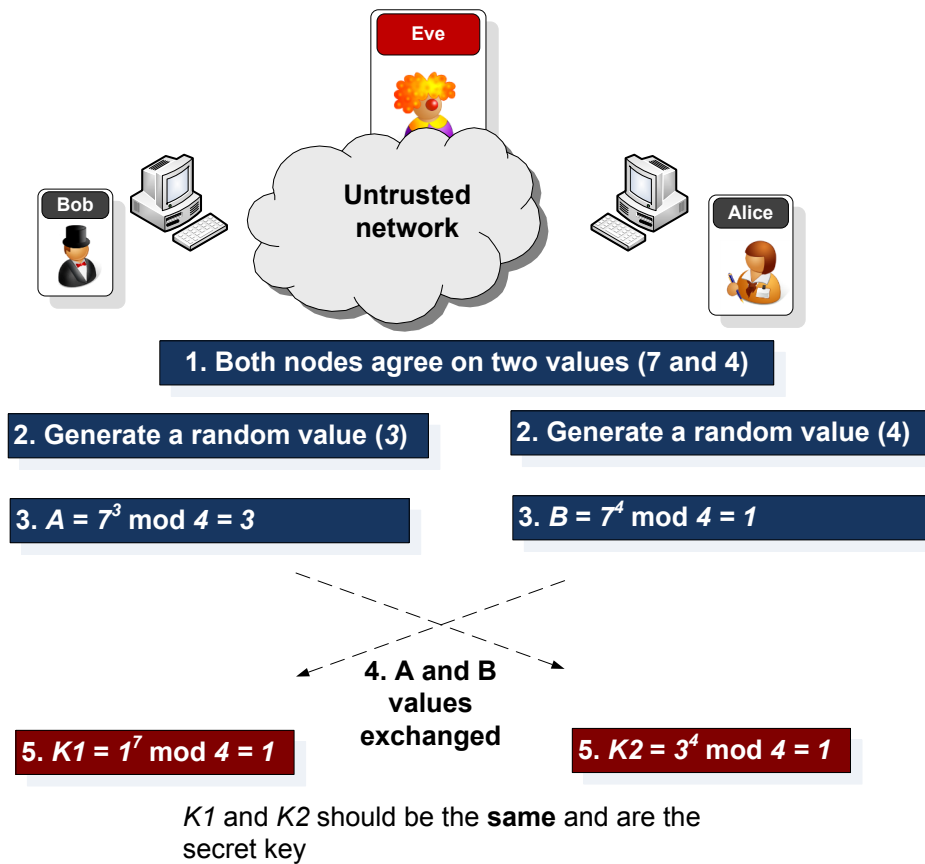


Figure 3.21 Example Diffie-Hellman process (see <http://buchananweb.co.uk/diffie.aspx>)

A simple .NET program to calculate small values of G and n is:

```
using System;
namespace diffie
{
    class Class1
    {
        public static Random r= new Random();

        static void Main(string[] args)
        {
            long x,y,A,B,n,G,K1, K2;

            G= 20;
            n= 99;

            x = random(10)+1;
            y = random((int)x);

            double val1= Math.Pow((double)G,(double)x);
            double val2= Math.Pow((double)G,(double)y);

            Math.DivRem((long)val1,n,out A);
            Math.DivRem((long)val2,n,out B);

            Math.DivRem((long)Math.Pow((double)B,(double)x),n,out K1);
            Math.DivRem((long)Math.Pow((double)A,(double)y),n,out K2);

            Console.WriteLine("x is {0} and A is {1}",x,A);
            Console.WriteLine("y is {0} and B is {1}",y,B);

            Console.WriteLine("K1 is: " + K1);
            Console.WriteLine("K2 is: " + K2);
            Console.ReadLine();
        }
        public static int random(int max)
```

```

    {
        try
        {
            return(r.Next(max));
        }
        catch {};
        return(0);
    }
}

```

which gives a sample run of:


```

x is 6 and A is 64
y is 3 and B is 80
K1 is: 91
K2 is: 91

```

It can be seen that the values of G and n (20 and 99, respectively) are relevantly small as larger values will typically overflow the calculations, as the `Math.DivRem()` method can only support long integers, whereas many more bits are required to support the large values involved, especially with the A and B to the power of x and y , respectively. A run of values of x and y between 1 and 3 shows that the values of $K1$ and $K2$ are the same for these values of G and n (the code for this is in Tutorial 3.13.5):

x	y	A	B	K1	K2
1	1	20	20	20	20
1	2	20	4	4	4
1	3	20	80	80	80
2	1	4	20	4	4
2	2	4	4	16	16
2	3	4	80	64	64
3	1	80	20	80	80
3	2	80	4	64	64
3	3	80	80	71	71

 **Web link:** <http://buchananweb.co.uk/security02.aspx> [Diffie-Hellman demo]

3.11 Public-key encryption

Public-key encryption uses two keys: a public one and a private one (Figure 3.22). These are generated from extremely large prime numbers, as a value which is the product of two large prime numbers is extremely difficult to factorize. The two keys are generated, and the public key is passed to the other side, who will then encrypt data destined for this entity using this public key. The only key which can decrypt it is the secret, private key. A well-known algorithm is RSA, and can be used to create extremely large keys. Its stages are:

1. Select two large prime numbers, a and b (each will be roughly 256 bits long). The factors a and b remain secret and n is the result of multiplying them together. Each of the prime numbers is of the order of 10^{100} .
2. Next, the public-key is chosen. To do this a number e is chosen so that e and $(a-1) \times (b-1)$ are relatively prime. Two numbers are relatively prime if they have no

common factor greater than 1. The public-key is then $\langle e, n \rangle$ and results in a key which is 512 bits long.

- Next the private key for decryption, d , is computed so that:

$$d = e^{-1} \bmod [(a-1) \times (b-1)]$$

- The encryption process to ciphertext, c , is then defined by:

$$c = m^e \bmod n$$

- The message, m , is then decrypted with:

$$m = c^d \bmod n$$

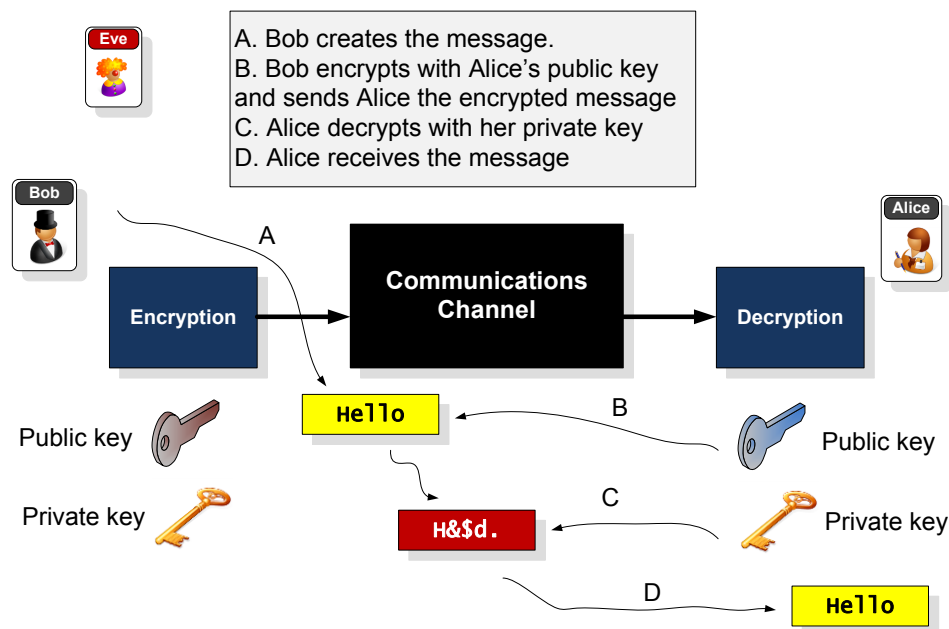


Figure 3.22 Public-key encryption/decryption process

Web link: <http://buchananweb.co.uk/security18.aspx> [Demo of RSA key gen.]

3.11.1 XML keys

The following is some .NET code to generate 1024-bit public and private keys:

```
System.Security.Cryptography.RSACryptoServiceProvider RSAProvider;
RSAProvider = new System.Security.Cryptography.RSACryptoServiceProvider(1024);
publicAndPrivateKeys = RSAProvider.ToXmlString(true);
justPublicKey = RSAProvider.ToXmlString(false);
StreamWriter fs = new StreamWriter("c:\\public.xml");
fs.Write(justPublicKey);
fs.Close();
fs = new StreamWriter("c:\\private.xml");
fs.Write(publicAndPrivateKeys);
fs.Close();
```

It converts them in an XML format, such as given in Figure 3.21 (which contains both the private and public key). In this case, the public key is:

```
<RSAKeyValue>
  <Modulus>
    1ntbP2f+I/3AiwKd+QeHhhsn1TkfufLKS4muFruJ8CwIRFhsyo9yoCIVydb6v0vDdtfg3
    F10iTGQw6waXy4QQ2LB4utIqASRumqu2cVNBLyKb/p7eHByTm3GAhxvyTOGWPidcbvCrIryor
    9ck9M79syetG7ZEpHd8hy4Qm6BuP8=
  </Modulus>
  <Exponent>AQAB</Exponent>
</RSAKeyValue>
```

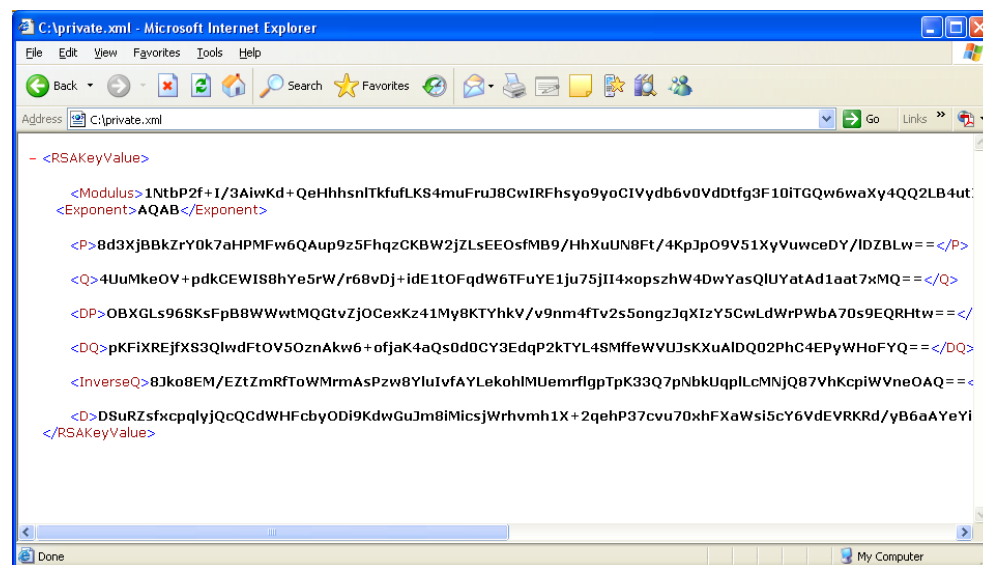


Figure 3.23 XML-based private key

The code to then read the keys is:

```
XmlTextReader xtr = new XmlTextReader("c:\\private.xml");
publicAndPrivateKeys=""; // reset keys
justPublicKey="";
while (xtr.Read())
{
  publicAndPrivateKeys += xtr.ReadOuterXml();
}
xtr.Close();
xtr = new XmlTextReader("c:\\public.xml");
while (xtr.Read())
{
  justPublicKey += xtr.ReadOuterXml();
}
xtr.Close();
```

and then to encrypt a message (txt) with the public key:

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
string txt= tbTxtEncrypt.Text;
rsa.FromXmlString(justPublicKey);
byte[] plainbytes = System.Text.Encoding.UTF8.GetBytes(txt);
byte[] cipherbytes = rsa.Encrypt(plainbytes,false);
this.tbTxtEncrypted.Text=Convert.ToBase64String(cipherbytes);
```

and then to decrypt with the private key:

```
RSACryptoServiceProvider rsa = new RSACryptoServiceProvider();
string txt=tbTxtEncrypted.Text;
rsa.FromXmlString(publicAndPrivateKeys);
byte[] cipherbytes = Convert.FromBase64String(txt);
byte[] plainbytes = rsa.Decrypt(cipherbytes,false);
System.Text.ASCIIEncoding enc = new System.Text.ASCIIEncoding();
this.tbTxtDecrypt.Text = enc.GetString(plainbytes);
```

where `tbTxtEncrypted` is the text box for the encrypted text, and `tbTxtEncrypt` is the text box for the text to be encrypted text. Using these keys, a message of “hello” becomes:

```
tPGI0dMBhQdwMNdN2hf/r1WkYsshK4rmfoshIdnwsiknw4ZLOtmc
gx3tuhoY3SNNP/z40ziigHUEcyp7POyYEPmAubC5XZmJZQCHKG+
3m2W1woAB09H4GxXK2P4q2BR61gekHoZOjyEMu2Bk7lCtiWYzPv9
gnubF7JWvfEuYmU=
```

Public-key encryption is an excellent method of keeping data secure, but it is often too slow for real-time communications. Also, we have the problem of distributing the public key to the sender. This problem is solved in the next unit by the use of digital certificates.

📖 **Web link:** <http://buchananweb.co.uk/security08.aspx> [RSA for ASP.NET]

📖 **Web link:** <http://buchananweb.co.uk/security16.aspx> [RSA for Windows]

3.12 One-way hashing

The concept of one-way hashing will be discussed in more details in the next chapter. One-way hashes are used for digital fingerprints and for secure password storage. Typical methods are NT hash, MD4, MD5, and SHA-1, and are used to convert plaintext into a hash value (Figure 3.24). It has applications in storing passwords, such as in Unix/Windows and on Cisco devices (Figure 3.25). A weakness of one-way hashing is that the same piece of plaintext will result in the same ciphertext (unless some salt is applied). Thus it is possible for an intruder to generate a list of hash values for a standard dictionary (Figure 3.26), and possibly determine the plaintext which makes the one-way hash. A major factor with hash signatures is:

- **Collision.** This is where another match is found, no matter the similarity of the original message. This can be defined as a Collision attack.
- **Similar context.** This is where part of the message has some significance to the original, and generates the same hash signature. This can be defined as a Pre-image attack.
- **Full context.** This is where an alternative message is created with the same hash signature, and has a direct relation to the original message. This is an extension to a Pre-image attack.

In 2006, for example, it was shown that MD5 can produce a collision within one minute, whereas it was 18 hours for SHA-1.

📖 **Web link:** <http://buchananweb.co.uk/security03.aspx> [MD5/SHA-1]

📖 **Web link:** <http://buchananweb.co.uk/security03a.aspx> [MD5/SHA-1]

📖 **Web link:** <http://buchananweb.co.uk/security03b.aspx> [MD5/SHA-1 with salt]

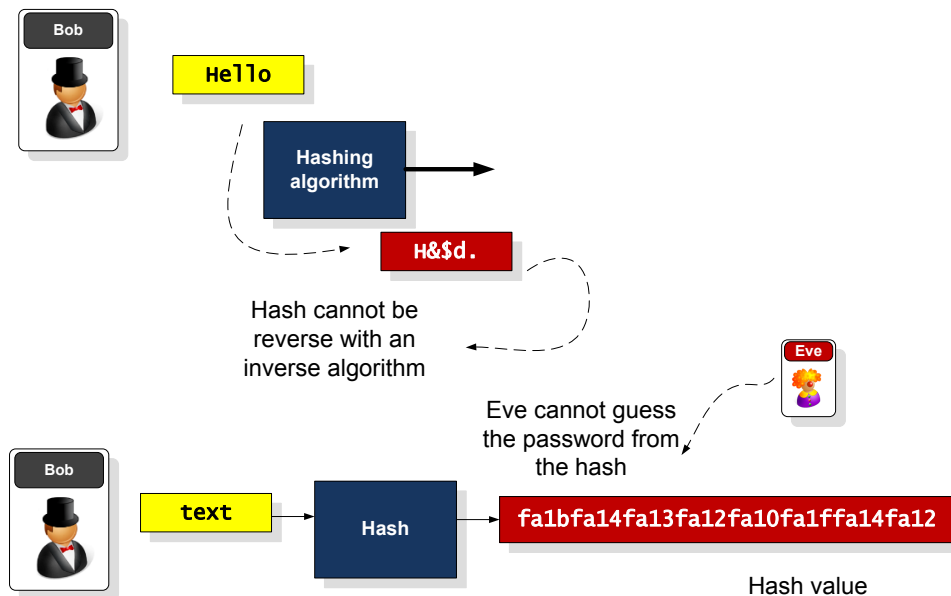
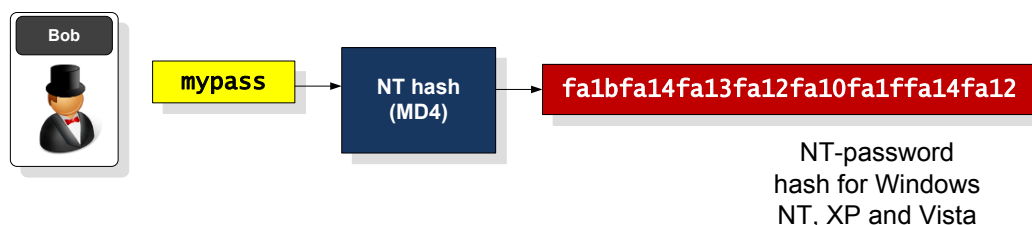


Figure 3.24 One-way hashing

Windows login/ authentication



Cisco password storage (MD5)

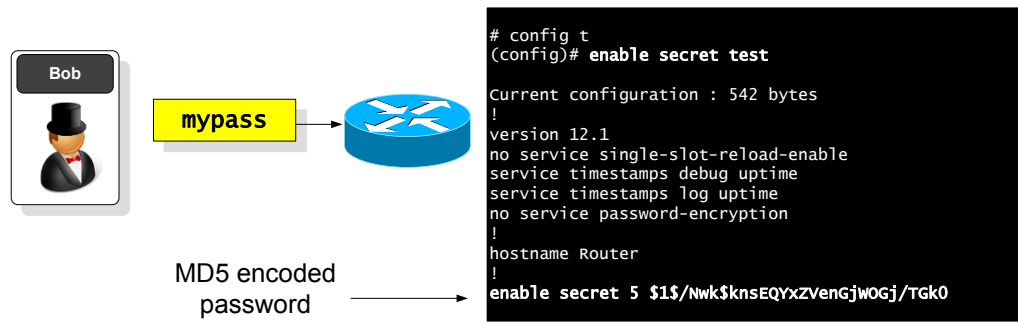


Figure 3.25 Application of one-way hashing

Windows login/ authentication

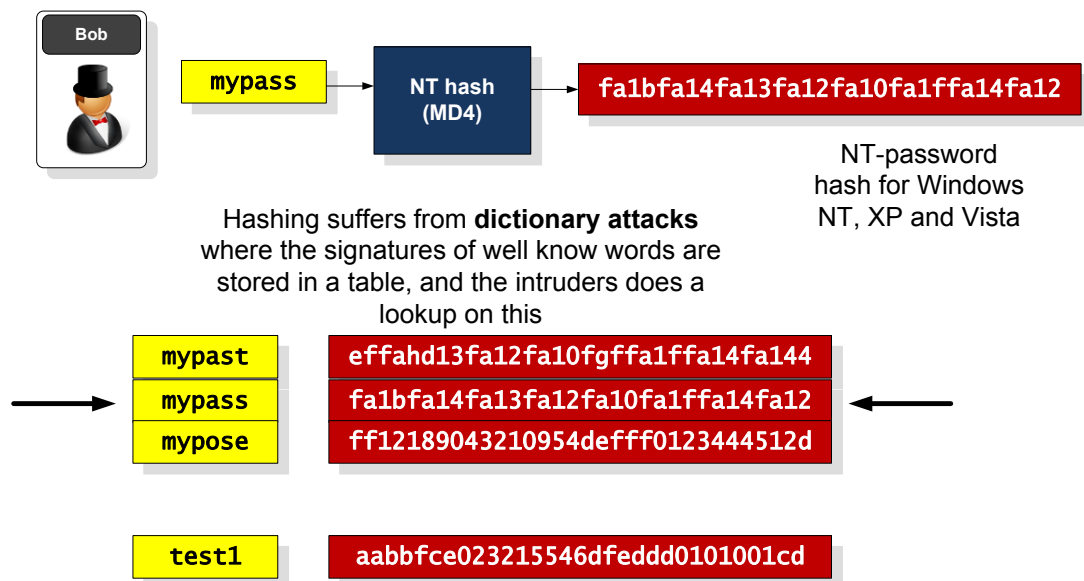


Figure 3.26 Application of one-way hashing

3.13 Key entropy

Encryption key length is only one of the factors that can give a pointer to the security of the encryption process. Unfortunately most encryption processes do not use the full range of keys, as the encryption key itself is typically generated using an ASCII password. For example in wireless systems typically use a pass phase to generate the encryption key. Thus for 64-bit encryption, only five alphanumeric characters (40-bits) are used and 13 alphanumeric characters (104 bits) are used for 128-bits encryption⁶. These characters are typically defined from well-know words and phases such as:

Nap1

Whereas 128-bit encryption could use:

NapierStaff1

Thus, this approach typically reduces the number of useable keys, as the keys themselves will be generated from dictionaries, such as:

About
Apple
Aardvark

⁶ In wireless, a 64-bit encryption key is actually only a 40 bit key, as 24 bits is used as an initialisation vector. The same goes for a 128-bit key, where the actual key is only 104 bits.

and keys generated from strange pass phrases such as:

xyRg54d
io2Fddse

will not be common (and could maybe be checked if the standard dictionary pass phrases did not yield a result).

Entropy measures the amount of unpredictability, and in encryption it relates to the degree of uncertainty of the encryption process. If all the keys in a 128-bit key were equally likely, then the entropy of the keys would be 128 bits. Unfortunately, due to the problems of generating keys through pass phrases the entropy of standard English can be less than 1.3 bits per character, and it is typically passwords at less than 4 bits per character. Thus for a 128-bit encryption key in wireless, and using standard English gives a maximum entropy of only 16.9 bits (1.3 times 13), which is equivalent, almost to a 17-bit encryption key length. So rather than having 202,82,409,603,651,670,423,947,251,286,016 (2^{104}) possible keys, there is only 131,072 (2^{17}) keys.

As an example, let's say an organisation uses a 40-bit encryption key, and that the organisation has the following possible phases:

Napier, napier, napier1, Napier1, napierstaff, Napierstaff, napierSoc, napierSoC, SoC, Computing, DCS, dcs, NapierAir, napierAir, napierair, Aironet, MyAironet, SOCAironet, NapierUniversity, napieruniversity, NapierUni

which gives 20 different phases, thus the entropy is equal to:

$$\begin{aligned} \text{Entropy(bits)} &= \log_2(N) \\ &= \log_2(20) \\ &= \frac{\log_{10}(20)}{\log_{10}(2)} \\ &= 4.3 \end{aligned}$$

Thus the entropy of the 40-bit code is only 4.3 bits.

Unfortunately many password systems and operating systems such as Microsoft Windows base their encryption keys on **pass-phrases**, where the private key is protected by a password. This is a major problem, as a strong encryption key can be used, but the password which protects it is open to a dictionary attack, and that the overall entropy is low.

3.14 File encryption

See on-line lecture

3.15 Tutorial

- 3.14.1** How many keys, in total, are used in the public-key system:
(a) 1 (b) 3
(c) 2 (d) 4
- 3.14.2** A typical public-key system is:
(a) IDE (b) IDA
(c) PGP (d) IDEA
- 3.14.3** How many keys, in total, are used in the private-key system:
(a) 1 (b) 3
(c) 2 (d) 4
- 3.14.4** A typical private-key system is:
(a) IDE (b) IDA
(c) PGP (d) IDEA
- 3.14.5** How many possible keys are there with a 16-bit key:
(a) 16 (b) 65,536
(c) 256 (d) 4,294,967,296
- 3.14.6** How many possible keys are there with a 32-bit key:
(a) 32 (b) 1,048,576
(c) 1024 (d) 4,294,967,296
- 3.14.7** If it takes 10ns (10×10^{-9} s) to test a key, determine the amount of time it would take, on average, to decrypt a message with a 32-bit key:
(a) 21.48 seconds (b) 43 seconds
(c) 21.48 minutes (d) 43 minutes
- 3.14.8** Which key does the recipient use to decrypt the main message:
(a) Recipient's public key (b) Recipient's private key
(c) Sender's public key (d) Sender's private key
- 3.14.9** Which key does the recipient use to authenticate the sender:
(a) Recipient's public key (b) Recipient's private key
(c) Sender's public key (d) Sender's private key
- 3.14.10** What bitwise operator is used in encryption, as it always preserves the contents of the information:
(a) Exclusive-OR'ed (b) AND
(c) NOR (d) OR
- 3.14.11** What happens when a bit-stream is Exclusive-OR'ed by the same value, twice:
(a) Bit-stream becomes all 0's (b) Bit-stream becomes all 1's
(c) Same bit-stream results (d) Impossible to predict

3.14.12 If it takes 100 days to crack an encrypted message, and assuming that computing speed increases by 100% each year, determine how long it will take to crack the message after two years:

- (a) 25 days (b) 44.44... days
(c) 50 days (d) 100 days

3.14.13 If it takes 100 days to crack an encrypted message, and assuming that computing speed increases by 50% each year, determine how long it will take to crack the message after two years:

- (a) 25 days (b) 44.44... days
(c) 50 days (d) 100 days

3.14.14 If there are only 1024 different passwords for a 64-bit encryption key, what is the key entropy [Hint: Key Entropy = $\log_2(X) = \log_{10}(X) / \log_{10}(2)$]

- (a) 1024 bits (b) 10 bits
(c) 64 bits (d) 18,446,744,073,709,551,616 bits

3.14.15 If there are only 4000 different passwords for a 64-bit encryption key, what is the key entropy:

- (a) 64 bits (b) 11 bits
(c) 11.97 bits (d) 12.2 bits

3.14.16 Using the following link for RSA encryption:

<http://buchananweb.co.uk/security18.aspx>

Enter a value of $p=11$, $q=3$. Prove that n becomes 33, ϕ is 20, and e could be 9, 13, 17, and so on. Keep pressing the $[e, n, \phi]$ button to regenerate a new value of e . Why does n and ϕ stay the same but e change?

3.14.17 Using the following link for Diffie-Hellman:

<http://buchananweb.co.uk/security02.aspx>

Enter a value of $G=40$, $N=10$, Bob $X=7$ and Alice $Y=10$, and prove that the resultant key are the same.

3.14.18 Using the following link for Diffie-Hellman:

<http://buchananweb.co.uk/security02.aspx>

Determine the shared keys for the following (the first one has already been completed):

G	n	Bob(x)	Alice(y)	Shared-key
15	58	6	7	57
16	58	7	5	
8	52	10	11	

3.14.19 Explain why public-key methods tend to be more secure than private-key methods. The discussion should include:

- Ease of changing the key.
- Ease of distribution.
- Crackability.
- etc.

3.14.20 Show that it will take 5849 years to search all the keys for a 64-bit encryption key. Assume it takes 10ns (10×10^{-9} s) to test a key. How might this time be drastically reduced?

3.14.21 If it currently takes 1 million years to decrypt a message then complete Table 3.7, assuming a 40% increase in computing power each year.

Table 3.7 Time to decrypt a message assuming an increase in computing power

<i>Year</i>	<i>Time to decrypt (years)</i>	<i>Year</i>	<i>Time to decrypt (years)</i>
0	1 million	10	
1		11	
2		12	
3		13	
4		14	
5		15	
6		16	
7		17	
8		18	
9		19	

3.14.22 The following messages were encrypted using the code mapping:

Input: abcdefghijklmnopqrstuvwxyz
 Encrypted: mgqoafzbcdiehjklnqrwsuvy

- qnv#mxo#oaqjoa#qbct#hattmza
- zjjogva#mxo#fmnasae#jxa#mxo#mee
- oaqjoa#qbct#mx#vjr#bmwa#fcxctbao#qbct#lratqcjx

Decrypt them and determine the message. (Note that a '#' character has been used as a SPACE character.)

3.14.23 The following messages were encrypted using a shifted alphabet. Decrypt them by determining the number of shifts. (Note that a '#' character has been used as a SPACE character.)

- (i) XLMW#MW#ER#IBEQTPi#XIBX
- (ii) ROVZ#S#KW#NBYGXSXQ#SX#DRO#COK
- (iii) ZVOKCO#MYWO#AESMU#WI#RYECO#SC#YX#PSBO
- (iv) IJ#D#YJ#IJO#RVIO#OJ#BJ#OJ#OCZ#WVGG

3.14.24 The following text is a character-mapped encryption. Table 3.1 defines the table of letter probabilities, and can be compared with the probabilities in the encrypted text.

tzf hbcq boybqtbmf ja ocmctbe tfqzqjejm v jyfl bqbejmr f cn tzbt ocmctbe ncmqben blf efnn baafqtfo gv qjcnf. bq v rqbwtfo ocntjltcj boof o tj b ncmqbe cn ofnqlcgfo bn qjcnf. tzc n qjreo gf mfgflbtfo gv futflqbe firckhfqt kljorqcm bclgjlf ntbtcq, aljh jtztfl ncmqben qjrkeqcm cqtj tzf ncmqbe'n kbtz (qljnn-tbed), aljh wctzcq fefqtlcqbe qjhkfqtgn, aljh lfqjlocq b qo kebvgbq hfocb, bq o nj jq. b qjhkblbtjl jrtkrtn b zcmz efyfe ca tzf ncmqbe yjetbmf cn mlfbtfl tz bq tzf tzlnzje o yjetbmf, fenf ct jrtkrtn b ejw. ca tzf qjcnf yjetbmf cn efnn tz bq tzf tzlnzje o yjetbmf tzfq tzf qjcnf wcee qjt baafqt tzf lfqjyflfo ncmqbe. fyfq ca tzf qjcnf cn mlfbtfl tz bq tzc n tzlnzje o tzflf blf tfqzqcirfn wzczq bq lforqf ctn faafqt. ajl fubhkef, futlb gctn qbq gf boof o tj tzf obtb fctzfl tj oftfqt fljl n jl tj qjllfqt tzf gctn cq fljl.

eb lmf bhjrqn ja ntjlbmf blf lfirclfo ajl ocmctbe obtb. ajl fubhkef, nfyfqtv hcqrtn ja zcac irbectv hrncq lfirclfn jyfl ncu zrqolfo hfmgtfn ja obtb ntjlbmf. tzf obtb jq qf ntjlf o tfqn tj gf lfecbgef bq wcee qjt ofmlbof jyfl tchf (futlb obtb gctn qbq benj gf boof o tj qjllfqt jl oftfqt bq fljl n). tvkcqbeev, tzf obtb cn ntjlf o fctzfl bn hbmqtfcq acfeon jq b hbmqtfcq ocnd jl bn kctn jq bq jkctqbe ocnd. tzf bqqlbqv ja ocmctbe nvntfhn ofkfqn jq tzf qrhgfl ja gctn rnf o ajl fbqz nbhkef, wzflbn bq bqbejmr f nvntfhn' n bqqlbqv ofkfqn jq qjhkfqt tjeflbqf. bqbejmr f nvntfhn benj kljorqf b ocaafclcm lfnkjnf ajl ocaafclqt nvntfhn wzflbn b ocmctbe nvntfhn zbn b ofkfqbgef lfnkjnf.

ct cn yflv ocaacqret (ca qjt chkjnncegf) tj lfqjyfl tzf jlcmqbe bqbejmr f ncmqbe batfl ct cn baafqtfo gv qjcnf (fnkfqbcev ca tzf qjcnf cn lbqojh). hjnt hftzjon ja lforqcm qjcnf cqjeyf njhf ajlh ja acetflcm jl nhjtzcm ja tzf ncmqbe. b mlfbt boybqtbmf ja ocmctbe tfqzqjejm v cn tzbt jq qf tzf bqbejmr f obtb zbn gffq qjyfltf o tj ocmctbe tzfq ct cn lfcbtcfv fbnv tj ntjlf ct wctz jtztfl krlf v ocmctbe obtb. jq qf ntjlf o cq ocmctbe ct cn lfcbtcfv fbnv tj kljqfn tzf obtb gfajlf ct cn qjyfltf o gbq cqtj bqbejmr f.

bq boybqtbmf ja bqbejmr f tfqzqjejm v cn tzbt ct cn lfcbtcfv fbnv tj ntjlf. ajl fubhkef, ycof bq brocj ncmqben blf ntjlf o bn hbmqtfcq acfeon jq tbkf bq b kctrflf cn ntjlf o jq kzjtjmlbkzcq kbklf. tzfnf hfocb tfqo tj bo o qjcnf tj tzf ncmqbe wzfq tzfv blf ntjlf o bq wzfq lfqjyflfo (nrqz bn tbkf zcn). rqa jlrqbtfev, ct cn benj qjt kjnncegf tj oftfqt ca bq bqbejmr f ncmqbe zbn bq fljl cq ct.

3.14.25 The following is a piece of character-mapped encrypted text. The common 2-letter words in the text are:

to it is to in as an

and the common 3-letter words are:

for and the

ixq rneq ja geie bjhrqtbeiqjtn etg bjhkriq wqisjwn qn qyq wqbwqenqtc. qi qn jtg ja ixq aq iqbxtjmcqem ewqen sqbx fwqtn fqtqain ij hni ja ixq bjrtiwqen etg ixq kqjkmqn ja ixq sjwmg. sqixjri qi hetv qtgrniwqen bjrmg tji quqni. qi qn ixq jfdabiyya ja ixq fjz ij qgnbrnn geie bjhrqtbeiqjtn qt e wqegefmq ajwh ixci nirgqtin etg kwjaqnnqjtemn emm jyq ixq sjwmg bet rtgqwnietg.

qt ixq keni, hni qmbiwjtb bjhrqtbeiqjtn nvniqhn iwethqaiqg etemjcrq nqctem. jt et etemjcrq iqmqkxjtg nvniq ixq ymieq mpyq awjh ixq kxjtg yewqen sqix ixq yjqbq nqctem. rtsetiqg nqctem awjh quiqwtem njrwbqn qenqmv bjwarki ixq nqctem.

qt e gqcqiem bjhhrtqbeiqjt nvniqh e nqwqgn ja gqcqiem bjgqn wqkwqnqtin ixq etemjcrq nqctem. ixqng ewq ixqt iwetnhqiig en jtqn etg oqwjn. gqcqiem qtajwheiqjt qn mqnn mqzqmv ij fq eaaqbiqg fv tjqnq etg xen ixrn fqbjhq ixq hjni kwqgjhteti ajwh ja bjhhrtqbeiqjtn.

gqcqiem bjhhrtqbeiqjt emnj jaaqwn e cwqeiqw trhfqw ja nqwyqbqn, cwqeiqw iweaaqb etg emmsn ajw xqcx nkqgg bjhhrtqbeiqjtn fqisqgt gqcqiem qlrqkhqti. ixq rnecq ja gqcqiem bjhhrtqbeiqjtn qtbmrgqn befmg iqmqyqnqjt, bjhkriqw tqisjwzn, aebnqhmq, hjfqmq gqcqiem wegqj, gqcqiem ah wegqj etg nj jt.

3.16 Software Tutorial

- 3.15.1** Prove that the following program can decrypt an encrypted message with the correct encryption key, while an incorrect one does not. Change the program so that the user enters the encryption key, and also the decryption key:

```
using System;
using XCrypt;
// Program uses XCrypt library from http://www.codeproject.com/csharp/xcrypt.asp
namespace encryption
{
    class MyEncryption
    {
        static void Main(string[] args)
        {
            XCryptEngine xe = new XCryptEngine();
            xe.InitializeEngine(XCryptEngine.AlgorithmType.TripleDES);
            xe.Key = "MyKey";


            Console.WriteLine("Enter string to encrypt:");
            string inText = Console.ReadLine();

            string encText = xe.Encrypt(inText);

            xe.Key = "test"; // should not be able to decrypt as the key differs

            try
            {
                string decText = xe.Decrypt(encText);

                Console.WriteLine("Input: {0}\r\nEncr: {1}\r\nDecr: {2}",
                                inText, encText, decText);
            }
            catch { Console.WriteLine("Cannot decrypt"); } ;
            Console.ReadLine();
        }
    }
}
```

 **Web link:** http://buchananweb.co.uk/srcSecurity/tut4_1.zip

- 3.15.2** The following program uses a single character as an encryption key, and then searches for the encryption key, and displays it. Modify it so that it implements a 2-character encryption key, and then a 3-character one:

```
using System;
using XCrypt;
// Program uses XCrypt library from http://www.codeproject.com/csharp/xcrypt.asp
namespace encryption
{
    class MyEncryption
    {
        static void Main(string[] args)
        {
            XCryptEngine xe = new XCryptEngine();
```

```

xe.InitializeEngine(XCryptEngine.AlgorithmType.TripleDES);
xe.Key = "f";
Console.WriteLine("Enter string to encrypt:");
string inText = Console.ReadLine();

string encText = xe.Encrypt(inText);
for (char ch = 'a'; ch <= 'z'; ch++)
{
    try
    {
        xe.Key = ch.ToString();
        string decText = xe.Decrypt(encText);
        if (inText == decText) Console.WriteLine("Encryption key found {0}",
                                                xe.Key);
    }
    catch {} ;
}
Console.ReadLine();
}
}
}

```

An example test run is:

```

Enter string to encrypt:
test
Encryption key found f

```


 **Web link:** http://buchananweb.co.uk/srcSecurity/tut4_2.zip

3.15.3 The following program implements a basic alphabet shifter. Modify it so that it implements a coding mapping system:

```

using System;
namespace alpha
{
    class AlphaShift
    {
        static void Main(string[] args)
        {
            string output = "defghijklmnopqrstuvwxyzabc";
            Console.Write("Enter a word to convert: ");
            string ins = Console.ReadLine();
            char [] inp = ins.ToCharArray();
            char [] oup = output.ToCharArray();
            Console.Write("Converted text is: ");
            for (int i=0; i<ins.Length; i++)
            {
                Console.Write(oup[inp[i]-'a']);
            }
            Console.ReadLine();
        }
    }
}

```

 **Web link:** http://buchananweb.co.uk/srcSecurity/tut4_3.zip


3.15.4 Modify the program in 3.15.3 so that it decodes the encoded output.

3.15.5 The following program uses Diffie-Hellman values of G of 20, and n of 99, for values of x and y from 1 to 4 and gives a sample run of:

x	y	A	B	K1	K2
1	1	20	20	20	20

1	2	20	4	4	4
1	3	20	80	80	80
2	1	4	20	4	4
2	2	4	4	16	16
2	3	4	80	64	64
3	1	80	20	80	80
3	2	80	4	64	64
3	3	80	80	71	71

```
using System;
namespace diffie
{
    class Diffie
    {
        static void Main(string[] args)
        {
            long x,y,A,B,n,G,K1, K2;
            G= 20;          n= 99;
            Console.WriteLine("x\ty\tA\tB\tK1\tK2");
            for (x=1;x<4;x++)
                for (y=1;y<4;y++)
                {
                    double val1= Math.Pow((double)G,(double)x);
                    double val2= Math.Pow((double)G,(double)y);
                    Math.DivRem((long)val1,n,out A);
                    Math.DivRem((long)val2,n,out B);
                    Math.DivRem((long)Math.Pow((double)B,(double)x),n,out K1);
                    Math.DivRem((long)Math.Pow((double)A,(double)y),n,out K2);
                    Console.WriteLine("{0}\t{1}\t{2}\t{3}\t{4}\t{5}",x,y,A, B, K1, K2);
                }
            Console.ReadLine();
        }
    }
}
```

 **Web link:** <http://buchananweb.co.uk/srcSecurity/diffie.zip>

Modify the program so that the maximum values of x and y are increased, and thus determine the maximum values that still produce the same values of K1 and K2.

3.17 Web Page Exercises

Implement following Web pages using Visual Studio 2008:

- 3.16.1 <http://buchananweb.co.uk/security07.aspx> [3DES]
- 3.16.2 <http://buchananweb.co.uk/security06.aspx> [RC2]
- 3.16.3 <http://buchananweb.co.uk/security15.aspx> [AES]
- 3.16.4 <http://buchananweb.co.uk/security18.aspx> [RSA]

3.18 Network Simulation Tutorial

- 3.17.1 On NetworkSims, go to **ISCW** and select Challenge 28. Create a policy, for the router, and determine the options:
 For encryption:
 For Hash:
 For Diffie-Hellman group:
 For Authentication method:
 Lifetime range (seconds):

An example is given next for the encryption options:

```
# config t
(config)# crypto isakmp enable
(config)# crypto isakmp policy 111
(config-isakmp)# encryption ?
  3des  Three key triple DES
  aes   AES - Advanced Encryption Standard.
  des   DES - Data Encryption Standard (56 bit keys).
```

Thus the options are 3DES, AES and DES.

3.17.2 On NetworkSims, go to **PIX_SNPA** and select Challenge 22. Create a policy, for the PIX firewall, and determine the options:

For encryption:

For Hash:

For Diffie-Hellman group:

For Authentication method:

Lifetime range (seconds):

3.19 Challenges

There are a number of challenges related to cipher coding. These are at:

<http://buchananweb.co.uk/security19.aspx> [ASCII coding]

<http://buchananweb.co.uk/security19a.aspx> [Bible codes]

<http://buchananweb.co.uk/security20.aspx> [Alphabet shifting]

<http://buchananweb.co.uk/security21.aspx> [Coded messages]

<http://buchananweb.co.uk/security22.aspx> [Covert channels]

<http://buchananweb.co.uk/security23.aspx> [Watermarks]

<http://buchananweb.co.uk/security25.aspx> [Test]

<http://buchananweb.co.uk/security26.aspx> [Scrambled alphabet]

<http://buchananweb.co.uk/security27.aspx> [Vigenère]

What was your final score?

Next complete: <http://buchananweb.co.uk/it4u00.aspx>

3.20 On-line Exercises

The on-line exercise for this chapter are at:

<http://buchananweb.co.uk/encryption.html>

3.21 NetworkSims Exercises

Complete:

Complete: PIX_SNPA Challenge I1-10

3.22 Chapter Lecture

View the lecture at:

<http://buchananweb.co.uk/security00.aspx>

and select **Principles to Encryption** [Link].
