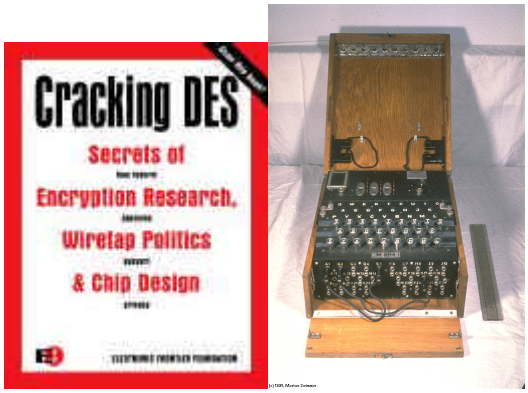
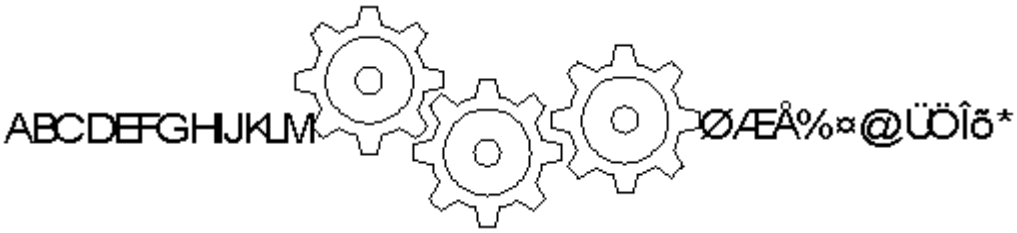
White Paper

ENCRYPTION





	February 18 1999
Classification	Public use
Author	Pascal LAMBERIEUX Norman ASA, Norway
Review	Sonja Draeger Norman Dev, Australia
Web version	Peter Maher Norman Dev, Australia

Contents

Getting started: Algorithms and keys

Contents

To scramble *plaintext* (original data) into *ciphertext* (the encrypted version) you need an *algorithm* and a key. The algorithm defines which encryption method is used, while the key cites a specific instance of the algorithm. The following example explains this further.

Simple example of an algorithm and key

Let’s take the following plaintext

To be or not to be..

and encrypt it to become

Up cf ps opu up cf

On first glance, the result appears incomprehensible. But since we have provided the plaintext and the corresponding ciphertext, it should be easy for you to deduce the algorithm and the key.

The solution? To encrypt the plaintext, replace each letter with the letter immediately following it in the alphabet. To decrypt the cyphertext, replace each letter with the letter immediately preceding it in the alphabet. The algorithm could be said to be "shifting the letters" or "rotation". The key is 1, because you shift (or rotate) 1 to the right.

Now try to decrypt this: *gb or be abg gb or*

This is a better encryption (cipher) of the same sort as the previous example. This is the ROT-13 cipher, which also rotates the letters as in the first example, but here the key is to shift every letter 13 positions in the alphabet. (This cipher was cracked in the time of Caesar.)

Restricted versus key-based algorithms

Contents

A *restricted-algorithm* algorithm is secure for as long as the way it works is kept a secret.

On the contrary, the security of a *key-based* algorithm does not rely on hiding the internals of the encryption mechanism, but on keeping the key used a secret. Even if the way the algorithm works is public knowledge, it can be used securely with a secret key.

Restricted algorithms are inadequate for a large group of users, such as in a large organization, because as soon as someone leaves the organization (or if someone accidentally reveals the secret of the algorithm) everyone else must switch to another algorithm. That is why mass market products only use *key-based* algorithms.

So, understand this paradox when dealing with *key-based* algorithms: they are considered secure **because everybody knows how they work**. If a vendor comes up with a home-made algorithm, there is no guarantee for the customer that it has no weakness or backdoor. The market prefers algorithms whose source have been published and that are trusted by the cryptography community.

Symmetric and public keys

Contents

Symmetric key algorithms

Symmetric-key encryption is an encryption method that uses the **same** key for encryption and decryption, as shown below. This type of encryption is **quick** and well suited when the data does not need to be shared (hard disk encryption, or file encryption of sensitive data on a PC).

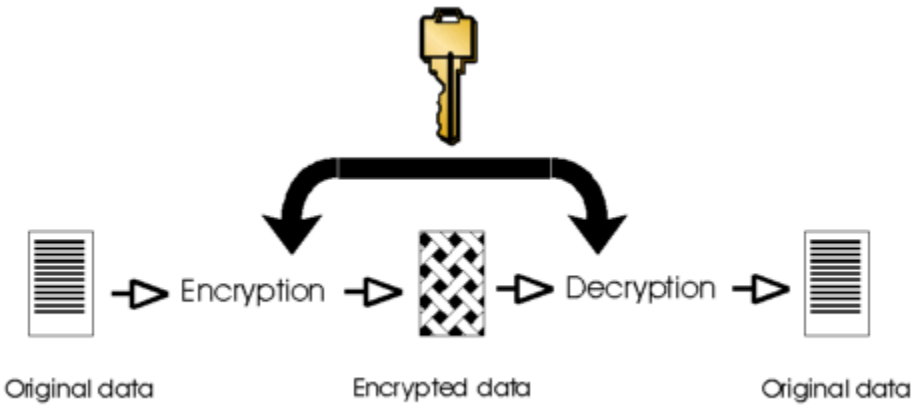


Fig 1: Symmetric key encryption

Well-known symmetric algorithms include:

- **DES** (Data Encryption Standard). Originally developed by IBM, DES was endorsed by the U.S. government in 1977 as an official standard. It has been extensively studied since its publication and is the most well-known and widely used cryptosystem in the world. Its reputation, however, has suffered from the frantic (and successful) cracking attempts of the cryptologic community (see Cracking an algorithm).
  DES uses a 56-bit key during encryption. The code is license free (public domain). DES has many variants (Triple DES, DESX).
- **IDEA** (International Data Encryption Algorithm) offers very good performance (twice as fast as DES) and high security. It is often considered as the quickest and most secure algorithm available to the public today. It uses a 128-bit key. The code is public, but commercial use is subject to license.
- **Blowfish**, written by Bruce Schneier, offers medium-level performance and high security. The key has a variable length from 32 to 448 bits. The code is license free.
- **RC5** (Rivest Cypher version 5) developed by RSA. The key length is variable (typical choices are 56 bits, 64 bits, and 128 bits). The code is public, but commercial use is subject to license.

Public key algorithms

A *public-key* algorithm (also called an *asymmetric* algorithm) uses a key pair. As shown below, the key used for decryption is **different** from the key used for encryption.

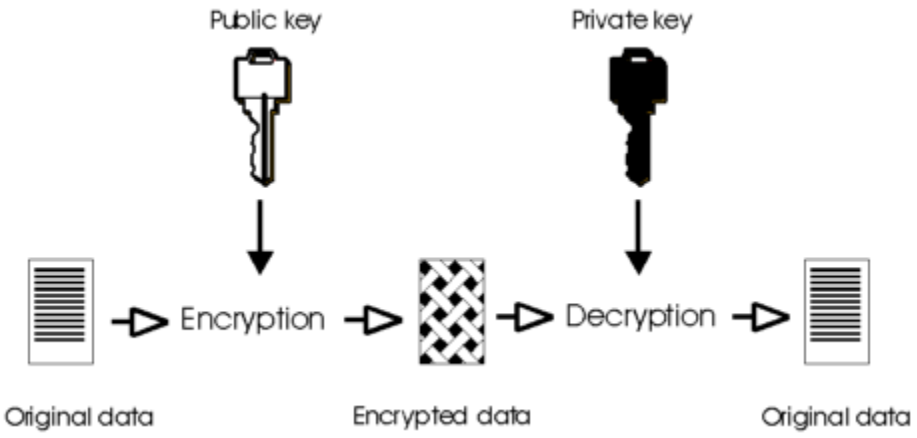


Fig 2: Public key encryption

In this system the **encryption** key can be made public (hence its name of *public key*) while the **decryption** key must be kept secret (*private* or *secret key*). This type of encryption is suitable when the encrypted data has to be shared or has to travel, for example, through e-mail, because there is no key to transmit so there is no risk of anyone intercepting it.

Let's take the example of a company using a public-key system for encrypting e-mail attachments. Every employee has his own pair of keys (public and private) plus a list of everybody's public key. When user A wants to send something to user B, he just looks up B's **public** key in the list, encrypts the file with it and sends it. User B then decrypts it with his **private** key. This is safe since only B's private key can decrypt the file and only B knows his private key.

This system works the two ways round: B can encrypt the file with his private key and send it to A. A can look up B's public key in the list and use it to decrypt the file. This does not give any confidentiality (since anybody can get B's public key), but proves to A that B is the real sender of the file (since he is the only one to know the private key). This technique is used for digital signatures.

The most popular asymmetric algorithm is RSA. RSA stands for Rivest, Shamir, and Adleman (its designers).

A mixture of both

The drawback of the public key system is the slowness of the encryption/decryption process. It makes it almost useless when processing big files. (In software, DES is about 100 times faster than RSA; in hardware 1000 times faster). To avoid this, a combination of public and symmetric keys can be used, as shown below:

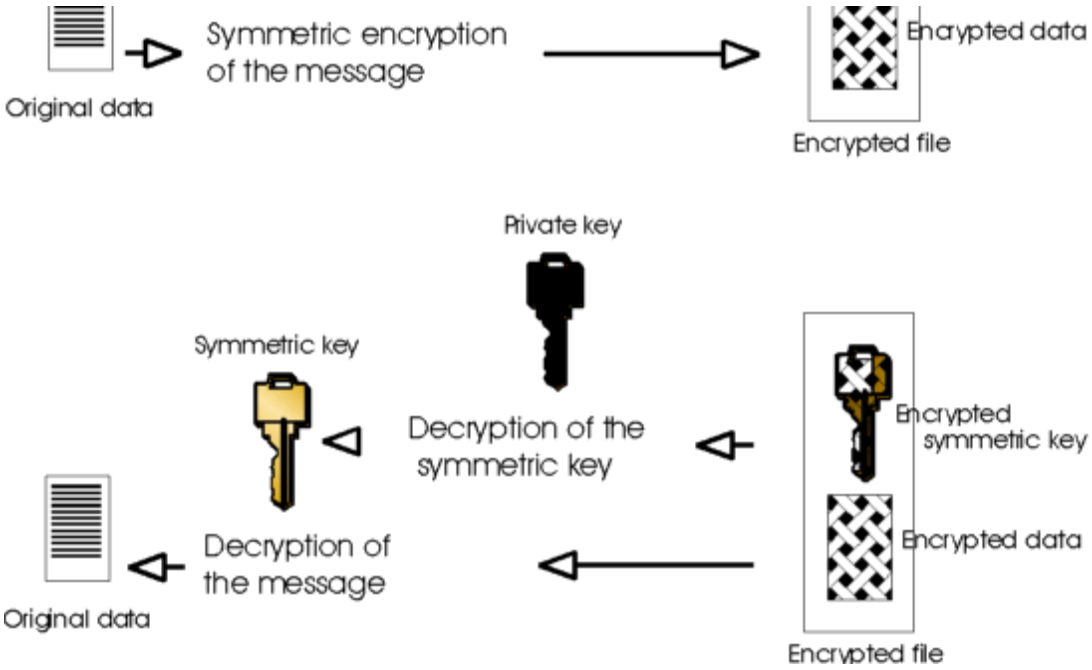


Fig 3: Mix of symmetric and Public key encryption

At encryption time, the plaintext is encrypted with a symmetric key. The symmetric key itself is then encrypted with the public key of the recipient, then stored at the end of the file.

At decryption time, the secret key is used to decrypt the symmetric key, which is then used to decrypt the message.

This technique is used, for example, by PGP (Pretty Good Privacy). PGP uses IDEA for the symmetric part and RSA for the public part.

Another Public key system widely used to safely distribute secret keys is the Diffie-Hellman protocol.

One-way hash functions

Contents

This is another interesting encryption system. One-way hash is a mathematical function that takes a string and scrambles it to a different (smaller) value **without** using any key. Its particularity is that it works in **only one direction**. Given a hash value, it is mathematically (almost) impossible to generate the original value.

The question is, then, "what is the advantage of encrypting something that you cannot decrypt at all ?". This technique is used when you want to check that somebody has exactly the same data as you. An example of how this is useful is using hashing to check the integrity of a message. You transmit the message plus the result of a hash function performed on your data. The recipient performs the same hash function on the message and compares the results. If the two hashed values match, it means that the original data has not been tampered with.

Another example of the use of one-way hashing is to handle a password check between a workstation and a server:

a) At password creation time, the user enters his new password. The password is hashed and the **hashed value** is sent to the server to be stored in the user database.
b) At logon time, the user enters his password. The password is hashed and the **hashed value** is sent to the server, which compares it with the one it has in its database. If they match, it means that the password entered by the user was correct.

Using this technique, the cleartext value of the password never travels on the network and is **not even stored anywhere**. In addition, there is no encryption key to manipulate so there is no risk of disclosure.

The most popular hash functions are MD5 and SHA.

Cracking an algorithm

Contents

Finding the plaintext of an encrypted message without knowing the key is called breaking, or cracking, an algorithm. It is very fashionable today to discuss the fact that such or such algorithm has been cracked. For example, DES and the 56-bits version of RC5 were cracked several times by a community of thousands of people worldwide during a contest organized by the American company RSA. Does it mean that DES and RC5 are weak? Not necessarily.

The important matter is not **if** an algorithm has been broken, but **how** it was broken. If it was cracked using a mathematical way or a backdoor, then it is weak.

If it was cracked using brute-force attack (which means to guess the key - see the next section) then we should consider the **resources** needed to achieve this. During a brute-force attack it took nearly nine months and several thousands of computers to find the right key for RC5. Recently, breaking DES went faster (22 hours in January 99), but almost 100000 PCs worked on it in parallel, including a supercomputer especially designed for that purpose.

If your encrypted customer file is stolen, do you think your competitors can afford putting this power on line to decrypt it?

Key length issues

Contents

Length=strength

Even if you assume that the algorithm you are using is perfect, you are still at risk of being a victim of a brute-force attack. This consists of trying all possible values of keys until the right one is found.

It is easy to demonstrate that, in the case of a brute-force attack, the security increases together with the length of the key. If the key is 8 bits long, there are 2^8 or 256 possible combinations. So it will take a maximum of 256 tries to find the right key. With a key length of 40 bits, you increase this to 2^{40}, namely a thousand billion combinations. The protection, then, resides on the time needed for a computer to successfully perform its attack.

The table below shows the average time for a hardware brute-force attack to succeed. (This concerns **only symmetric** key algorithms. For public-key algorithms, the problem is a bit different.)

Power / cost	40 bits (5 char)	56 bits (7 char)	64 bits (8 char)	128 bits (16 chars)
$ 2K (1 PC. Can be achieved by an individual)	1.4 min	73 days	50 years	10^{20} years
$ 100 K (This can be achieved by a company)	2 sec	35 hours	1 year	10^{19} years
$ 1 M (Achieved by a huge organization or a state)	0.2 sec	3.5 hours	37 days	10^{18} years

Fig 4: Estimate time for successful brute-force attack

Now, the real world: Below are the result of the RSA contest to break the various versions of RC5.

Key length	Started	Finished	Duration
40 bits	January 28 1997	January 28 1997	3,5 hours
56 bits	January 28 1997	October 20 1997	265 days
64 bits	January 28 1997	Still ongoing	-
128 bits	January 28 1997	Still ongoing	-

Fig 5: Status of the RSA RC5 contest as of February 1999

The next example shows how the time needed to crack an algorithm reduces as the power of the computers increases year after year. Below is the result of the three contests where DES was cracked:

Started	Duration
January 1997	96 days
February 1998	41 days
January 1999	22 hours

Fig 6: Results of the RSA DES contest 1997/98/99

Based on these figures, the common opinion today is that

- 40 bit key algorithms are useless,
- 56 key algorithms offer good privacy but are vulnerable,
- 64 bit algorithms are safe today but will be soon threatened as the technology evolves,
- 128 bit and over algorithms are almost unbreakable.

The point of view of the authorities

Since the key length parameter is a good indicator of the strength of an algorithm, it is commonly used in the different laws regulating the use of encryption.

But first, one can wonder why a government should bother to regulate the use of encryption. This is mostly for two reasons:

- because they consider encryption as a military weapon (World War Two proved the advantage of having an encryption system that the enemy could not break), and
- because they want the justice to be able to open any document if needed.

So in the first case the government will control the **export** of encryption tools, and in the second case, the **use** of them. (The control of **importing** is a different problem that mostly addresses economical issues).

The regulations are very different in the world and are constantly evolving in ways (as shown by the three examples below), from total freedom (Norway) to strict export control (USA) to an almost complete prohibition (France).

As of today (February 99) there is almost no limitation in Norway, so the happy Norwegians are free to use and export whatever encryption tools they want. (This could change with the Wassenaar agreement, which could lead to an export restriction of algorithms with keys greater then 56 or 64 bits).

The US has no limitation in the use of algorithms, but was prohibiting, until December 98, the export of keys greater than 40 bits. This has been, and is still, a real handicap for US companies selling encryption products. The limit has now been raised to 56 bits and is completely removed for some sectors of the economy, such as insurance and health care.

Since 1994, France has prohibited the **use** of keys greater than 40 bits, and the poor Frenchmen had to forget both confidentiality and most of the benefits of the Internet (since transactions could not be secured). To address this, the French government has promised to raise this spring (1999) the limit to 128 bits.

Encryption in Norman software

Contents

- message/file/directory encryption (Norman Privacy),
- hard disk encryption (Norman Access Control), and

- TCP/IP connection encryption (Norman Security Server).

The latter is not described in this document, but in the white paper "Internet Security".

Norman Privacy

Contents

Norman Privacy uses the symmetric algorithm Blowfish (see Symmetric and public keys) to encrypt

- part or all of the **contents** of a file,
- a file, and
- a directory or a floppy disk.

Encryption key

At encryption time, the user must provide a password, which is used as an encryption key. Blowfish allows a variable key length, so the length of the password (which can be from 1 to 56 characters) is the length of the key (which is equivalent to 8 to 448 bits).

The key is hashed using MD5 (see One-way hash functions) and stored in the header of the encrypted object (for messages or files) or in the registry (for directories). The clear text value of the password is not stored anywhere.

At decryption time, the user must provide the same password as the one that is hashed. The password is compared to the hashed value stored in the header or the registry. If the comparison is successful, decryption proceeds using the password provided by the user.

Profiles

Privacy uses a symmetric algorithm and was primarily designed to protect sensitive information that was not supposed to leave the PC. As said in section 3, if a file has to travel outside of the PC, then the public key system might be more appropriate.

Does it mean that you can’t use Privacy to encrypt e-mails? No. Privacy can be a good alternative for an organization that does not want to bother with the distribution and management of public keys. Privacy provides a simple solution: *User profiles*.

Profiles are encrypted files containing user passwords. To share your encryption key with other people, all you need to do is export your profile (which is then automatically encrypted) from Privacy and send it to them. The recipients must get from you over a secure channel (phone, fax) the password used to encrypt your profile, decrypt it, and import it into Privacy. After this, to decrypt the files you send them, they just need to click your name in a list and Privacy will automatically extract your key from the profile. This means that they will never directly manipulate your key; they will never even know it. The screen below shows how the list of profiles appears for the end-user:

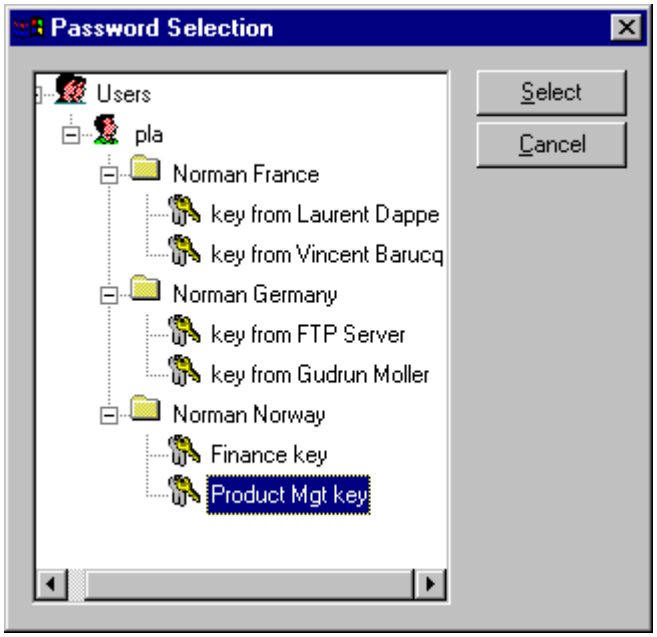


Fig 7: Privacy Profiles

Profiles have another advantage: **They can move with the user**. Privacy can store them on a **smart card** so that they are not stored on a specific PC but can be used on any machine.

Self decrypting files

Let's now address the following problem: How can I send an encrypted file to recipients outside the organization, knowing that they are not using Privacy and probably have no encryption software at all?

Privacy provides a convenient way to do this: it uses *self decrypting files*. At encryption time, Privacy generates a **program** file containing the encrypted data plus the minimum code required to decrypt it. The program file is sent to the recipient, while the key is transmitted by phone or fax.

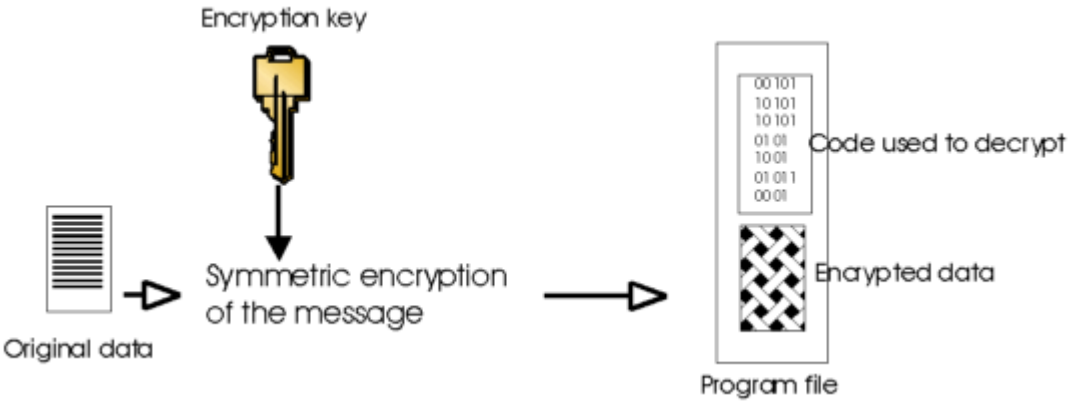


Fig 8: Building a self decrypting file.

The recipient simply runs the program to decrypt the file, as shown on the following dialog box.

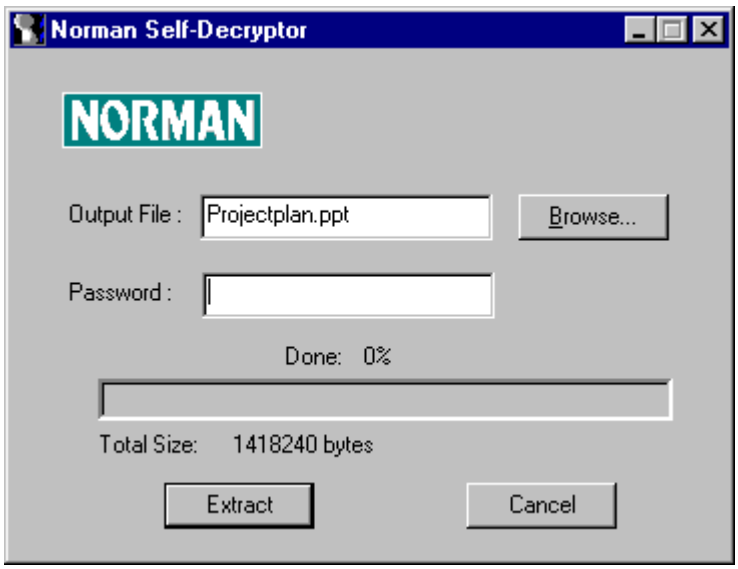


Fig 9: Self Decryption output screen

Norman Access Control

Contents

Norman Access Control protects your desktop against unauthorized access. It provides logon control, boot protection (these two features being described in the white paper on Access Control) and hard disk encryption.

While logon control and boot protection offer a fair protection level, they still can be circumvented by special tools that read the disk physically sector by sector. Hard disk encryption is the ultimate security level because it makes the disk completely unreadable for unauthorized persons.

Algorithms

Name	Key length (bits)
XOR	8-512
Blowfish	8-448
DES	56
DESX	128
IDEA	128

Fig 10: Algorithms available for Hard Disk encryption

All these algorithms are described in section 3, except XOR. Despite its key length, XOR is the **weakest** of the algorithms in the list. It is provided for low security, but maximum performance.

On multi-partition PCs it is possible to chose a different algorithm for every partition. For example,

one drive unencrypted, one encrypted with XOR, and one encrypted with Blowfish.

The disk encryption module allows a custom plug-in of other algorithms.

Encryption key

There are two ways to handle the encryption key:

- The key is hashed using MD5 and stored within reserved sectors of the disk.
- The key is not stored anywhere. In this case the user **must** enter the key at startup time. This is the most secure way, but the user cannot afford to forget the key because without it there is **no backdoor** and no other possible entry to the system.

Encryption process

The encryption time will vary depending on the algorithm used. The average is about 1 Gb per hour.

Norman will provide, in mid-99, a creeping encryption module. This module encrypts only the part of the disk containing data, which considerably reduces the encryption time.

During the encryption process the encryption module maintains a table indicating which record is currently being encrypted. This allows NAC to resume the process at the point it was stopped in case of abnormal termination (loss of power for example).

Decryption process

At boot time Norman Access Control will either ask the user to provide the encryption key, or retrieve the key automatically from the boot sector (depending on which option has been chosen - see previous section). Decryption is transparent to the user and done on the fly by a module loaded in memory that intercepts both the read calls (to decrypt the data) and the write calls (to encrypt the new data added to the disk).