CRYPTOLOGIA January 1987 Volume XI Number 1

A SURVEY OF DATA INSECURITY PACKAGES

Martin Kochanski

ABSTRACT: Five commercially available encryption packages for the IBM PC are described and broken.

KEYWORDS: Computer security, encryption, cryptanalysis, software.

INTRODUCTION

As people become more aware of the need for data security on personal computers, a rapidly increasing number of packages have appeared to meet this need. We ourselves entered this market in late 1984 with Ultralock.

From time to time we buy and evaluate other people's data security products for the IBM Personal Computer, either because they seem similar to Ultralock or because they look interesting in themselves. We have always concentrated on ease of use, practicality, and speed, and not on security as such. We spent a year working on and gradually refining our algorithm, and we assumed that anyone else who launched a security system would be just as careful. We were wrong. This article describes the algorithms we found, and how to break them.

Security packages vary widely in the features they provide. Some simply encrypt files on request; others provide elaborate password and access control schemes, and then use encryption to enforce them: because personal computers are such open systems, a password scheme without encryption is useless - all you need to do to break it is to insert a disk containing a copy of a less scrupulous operating system, or some clever direct disk reading utility.

Whatever the facilities provided, the strength of any PC security thus rests ultimately on the strength of its encryption algorithm.

TESTING SECURITY

We assume that someone attacking a system knows which protection program was employed, and that he can buy a copy of it.

We assume that he knows what sort of thing the files he is attacking contain: text, or data, or programs.

We assume that he has access to an ordinary PC, but to nothing more sophisticated.

In one case only, we assume a traditional known-plaintext attack: that the attacker has one "Rosetta Stone" file in both plain and encrypted versions. This is quite probable in a business environment:

Some documents eventually cease to be sensitive and are then no longer encrypted.

In a system where files are encrypted and decrypted in batches, a known plaintext can be inserted before encryption.

If some standard file - a word-processing format definition, or even a program file – is encrypted, the plaintext will be known anyway.

With electronic mail, the attacker may be able to send his victim a message knowing that it may be encrypted.

We do not assume that the attacker is capable of disassembling and analysing the program code of the protection system.

INHERENT AND USER KEYS

Several of the packages reviewed use inherent keys - that is, keys that are built into the code of the package itself and are different for each copy sold. Such packages are usually copy-protected, requiring the presence of a particular key disk before they can be run, and this disk is kept locked up when not in use. There are normally provisions for ordering multiple identical copies if necessary.

We do not assume that an attacker's copy of a package will have the same inherent key as the copy he is attacking.

With a well-designed cryptosystem with adequate key management, inherent keys are not really necessary. They have some use as a way of disguising the deficiencies of less secure systems.

THE TESTS

When we decided to test some security products, we looked for answers to the following questions:

Given the package, can one, by means of test encryptions, deduce the algorithm used?

Can one write a simple program which, given an encrypted text file with an unknown key, will decrypt the file? Programs are far less intelligent than people, and less good at inspired guessing, so we can be sure that if a program can decrypt a file all by itself, an attacker will be able to break the code too.

Given some plain text and corresponding encrypted text in a file, can one deduce a key that can be used to decrypt the rest of the file, or other files encrypted with the same key? As we have explained, this is most important in a business environment.

The breakability which we demonstrate for the cryptosystems we describe is not the usual rather rarefied breakability which requires only a week's time on a Cray. We use an ordinary IBM PC. All the breaking programs are quite small and simple, and even the slowest only takes a few minutes to run. The programs for Superkey, Padlock, and PS3 are written in assembler, because the breaking algorithms are simple; the programs for Crypt and N-Code are written in Turbo Pascal.

DEFINITIONS

We explain here various terms which are more familiar to computing specialists than to cryptographers. We also define some notational conventions.

A *bit* is a binary digit - its value is 0 or 1.

A *byte* is 8 bits. It has no inherent meaning, but acquires different meanings depending on how it is used. Here are some of the possible meanings:

- A sequence of 8 separate bits (formally, an element of $GF(2^8)$).
- A number between 0 and 255: the bit values simply form the binary representation of the number, so that 00001010 represents the value 10 (decimal).
- A character or a display control code. Each possible character has a unique pattern of 8 bits associated with it: for instance, the American Standard Code for Information Interchange (ASCII) associates the bit pattern 01000001 with the character "A".

Logical operations such as AND, OR, NOT, and XOR (exclusive-OR) interpret bytes as sequences of bits, and work on each of the 8 bits independently of the others; arithmetic operations such as addition and subtraction interpret bytes as numbers modulo 256; and the simplest output operations interpret bytes as characters. Most computer cryptographic algorithms work by interpreting the same byte in several different ways in the course of their operation: what was a displayable character is processed as if it were a number or a sequence of bits.

We shall frequently use *hexadecimal* (base-16) representation for bytes: the hexadecimal digits are 0 to 9 followed by A to F, and we shall prefix isolated hexadecimal numbers by a $ sign to distinguish them from decimal ones. For example, 41 = forty-one; $41 = 4 * 16 + 1 = sixty-five; $1A = 1 * 16 + 10 = twenty-six.

A *word* is here defined as 16 bits, with bit-sequence and numeric interpretations (from 0 to 65535) like those of bytes. In addition, a word is sometimes considered as a pair of bytes, in which case the first byte of a word value N has the (numeric) value N mod 256 and the second byte has the value N/256: so the word value $1ADC is treated as the byte $DC followed by the byte $1A. This "backwards" representation is a consequence of the way in which numbers are stored by the microprocessor.

Array subscripts are all assumed to start at zero.

"*Emphasised quotations*" are taken from the suppliers' product literature, documentation, and publicity material.

ASCII

Almost all software packages represent characters in ASCII, and so the cryptanalyst will often be faced with encrypted ASCII files. ASCII has some useful characteristics for cryptanalysis:

Codes from $00 to $1F (and also $7F) are *control codes*, and are almost unused in most files, except for $0D (carriage return) and $0A (line feed). $1A is often used as an end-of-file marker.

Codes from $20 (the space) to $7E are printable characters. Upper-case letters run from $41 to $5A, and lower-case ones from $61 to $7A. This leads to useful irregularities in the distribution of the values at certain bit positions.

Codes from $80 to $FF are undefined. The IBM PC uses these for special characters such as accented letters and mathematical symbols, and they are therefore rare or absent in most ASCII text.

Byte rotation

Several packages *rotate* bytes. Rotating a byte one bit to the right means removing the least significant bit from the bit sequence and placing it in the most significant bit position: so the bit sequence abcdefgh becomes habcdefg. Rotating n bits to the right just performs this process n times; rotating to the left is the same process reversed, so that abcdefgh becomes bcdefgha. For example, 01000001 rotated 3 bits to the right becomes 00101000.

SUPERKEY

Superkey is a combined keyboard enhancer and file encryption program.

At any time, even in the middle of a program, one can ask Superkey to encrypt or decrypt a file. Superkey has two encryption methods: DES, which, like all software implementations of DES, is slow (and in this case also unavailable outside the USA), and a fast proprietary algorithm which the supplier says "*uses our highly efficient proprietary algorithm for file encryption. It protects your file against any but the most sophisticated intruders.*"

There is no inherent key. The key consists of the name of the file being encrypted, added to a 1 to 30 character password entered by the user. The key is not stored anywhere once the file has been encrypted.

Deducing the algorithm

To deduce the algorithm, we encrypted a 32-kilobyte file, called ZERO.000, consisting entirely of zeros ($00), with the key "0". The encrypted text repeated itself every 1024 bytes. We repeated the whole process with a new copy of ZERO.000 with the first byte replaced by $01, and found that only one bit of the output was affected. A few more such tests satisfied us that the encryption algorithm was a straight byte-by-byte exclusive-OR with a repeated 1024-byte key stream.

We then encrypted an identical file, called it ZERO.100, and looked at the differences. Here are the first 64 bytes of each file, in hexadecimal:

ZERO.000

```
18 2D A2 29 A7 18 96 51 94 D3 0C A8 4A E9 06 25
F4 03 7A 81 C0 0C 4B 54 92 3D 60 A5 2A 49 9E 30
15 A4 4F 18 52 A7 0C D3 06 03 06 D2 8A 29 E9 81
69 45 94 F4 C0 A2 4A 7A 60 25 3D 30 9E 18 0C 03
```

ZERO.100

```
18 2D A2 29 A7 98 96 51 94 D3 4C A8 4A E9 26 25
F4 13 7A 89 C4 0C 4B 54 92 3D 62 A5 2A 49 9E 31
15 A4 4F 98 52 A7 4C D3 26 13 06 D2 8A 29 E9 89
69 45 94 F4 C4 A2 4A 7A 62 25 3D 31 9E 98 4C 03
```

The one-bit change in the filenames has caused a number of changes in the key stream, and these are underlined above. The first change is an exclusive-OR with $80, the second with $40, and so on; also, the changes start far apart, get closer together, and then repeat. We conducted similar experiments by changing various bits in the filename and in the key, and quite rapidly deduced the Superkey encryption algorithm:

The algorithm

First, construct a source key S by concatenating the user-entered key with the name of the file being encrypted and truncating the result to 28 bytes if it is longer. Only the first character of the filetype is used.

Next, construct a key stream. The process depends on the length s of S, but it is easiest to describe if we assume a particular key length: say s = 6 bytes.

- Form the 21-byte string S1 = 012345 12345 2345 345 45 5. (We have inserted spaces only to make the pattern clear). In general, S1 is s(s + 1)/2 bytes long.
- Repeat S1 until you have a string S2 128 bytes long, ignoring surplus bytes from the final occurrence of S1: in this case, this means 6 full copies of S1 and one partial copy.
- Repeat S2 8 times, giving 1024 bytes in all: call this sequence S3. S3 is therefore a sequence of 1024 entries, each between 0 and 5 (in general, between 0 and s - 1).
- Build the 1024-byte key stream K as follows:

```
For t := 0 to 1023
   j := S3[i] (j is an index into the source key S)
   Rotate S[j] right by one bit
   K[i] := S[j]
```

  Note that after 1024 steps – i.e. 8 iterations of S2 – each element of S will have been rotated a multiple of 8 times and will thus have returned to its original value; also that the later bytes in the source key S are used more often (e.g. "5" occurs far more often than "0" in S1) and so rotate faster than the earlier ones.

- Given a file containing plaintext bytes P, turn this into ciphertext C by:

```
C[i] := P[i] XOR K[i mod 1024]
```

The attack

"*An encrypted file contains absolutely no information about the encryption ... Borland cannot help you restore scrambled files.*"

The majority of bytes in text files are standard ASCII characters and thus do not have their top bit set. Even in a word processing package such as Wordstar, which sets the top bit of bytes to indicate the ends of words, the proportion of top bits set is never more than 25%.

If we knew the length of the source key S, we could look at each bit in S in turn. Because of the way the key stream is constructed, every bit will eventually be shifted into the top bit position.

For each bit b of S

- Find all the places j in K where b has been shifted into the top bit position.
- Look at the corresponding places in the ciphertext (i.e. C[i], for every i that equals some j modulo 1024), and count how many top bits are set.

- We know that hardly any of the top bits were set in the plaintext. If most of the top bits are set in this sample of ciphertext, they must have been inverted by the encryption, so b must be 1; if most of the top bits are not set, b must be 0.

Of course, the drawback is that we do not know the length of the source key S. So we try this attack assuming every key length from 1 to 30 bytes.

Assume that the key length is 6 and we are trying the attack with a key length of 11. Then when we look at a sample of ciphertext chosen to correspond to a particular bit b of our 11-byte S, we will actually be looking at a sample that corresponds to a mixture of bits from the real 6-byte S. Some of these bits in the real source key will be 0 and some will be 1, so our sample will be a mixture of plaintext bits (wherever the real key bit is 0) and inverted plaintext bits (wherever the real key bit is 1). So instead of being strongly skewed, as it would have been if the sample had corresponded to just one key bit, the distribution of top bits in our sample will be fairly evenly balanced.

To choose the key length, then, we look for the length which consistently gives the most extreme proportions of top bits. Scanning through all possible key lengths takes less than 5 seconds on an IBM PC.

We have written a small program which performs this attack on a given file. On a test word-processing disk, it decrypts all text files and most program files over 3,000 bytes in length; and decrypts all text files shorter than this (except for four under 130 bytes) with at most 1 character in 16 wrongly decoded: such files are still quite legible. These errors will appear as errors in the deduced key - for instance, CO^FIDENTIAL instead of CONFIDENTIAL - and the real key can usually be guessed. Otherwise, a more sophisticated version of the attacking program could accept the user's corrections, in the same sort of way as the attacking program for Crypt (described below).

PADLOCK

The recently-launched Padlock is a pure file encryption system. It is *"based on principles used currently in international espionage*".

To encrypt or decrypt a file, one uses separate commands ENCODE and DECODE: one cannot invoke Padlock from within an application program. The key for a file is built from:

an inherent key;

a key based on the time of day;

and an optional user-specified key up to 8 characters long: this is displayed on the screen throughout the encryption or decryption process.

An encrypted file has a header attached to it which contains the whole of the file key (encrypted, of course): a separate master disk is supplied to allow this header, and hence the whole file, to be decrypted if a user should forget the key he specified.

The algorithm

The Padlock encryption process is "*generally accepted by cryptologists as being unbreakable unless both the procedure and the cipher-key are known*".

The key stream K is constructed using a Fibonacci-type procedure, starting from three 16-bit seeds, K[0], K[1], K[2]: these are calculated from the file key by a procedure which we have not investigated. All manipulations are performed on 16-bit words, so all calculations are effectively done modulo 65536 (=$10000).

Build a 32766-byte key stream as follows:

For i := 3 to 16382

K[i] := K[i - 2] + K[i - 3]

Given a file containing plaintext words P, turn this into ciphertext C by:

C[i] := P[i] + K[i mod 16383] (mod 65536)

This is quite a good method for generating pseudo-random numbers K, but (as we shall see) pretty weak as a cryptosystem.

The algorithm was fairly easy to deduce. We started by establishing that encryption was done by modulo-65536 addition, and then simple visual inspection yielded the rule by which the key stream was constructed. The length of the key stream was deduced when our own decryption of a long file suddenly failed after 32766 bytes.

The attack

"*It would be theoretically possible for an expert to ... unscramble one of your encoded files. To do this he would require access to your individual copy of the PADLOCK modules as well as to the encoded file. It could not be done using any other copy of PADLOCK.*"

Given any three consecutive words of known plaintext in a known position, one can deduce three words of the key stream, and hence deduce the whole key stream

forwards: K[i] = K[i - 2] + K[i - 3]

and backwards: K[i] = K[i + 3] - K[i + 1]

Given any four consecutive words of known plaintext in an unknown position, one can search for them assuming each position in turn; deducing the key from three of them, and seeing whether the deduced key will yield the fourth word in the correct position.

For many attacks, it is worth eliminating the key. Compute:

C'[t] = C[i] + C[i + 1] - C[t + 3]

= P[i] + K[i] + P[i + 1] + K[i + 1] - P[i + 3] - K[i + 3]

= P[i] + P[i + 1] - P[i + 3] + K[i] + K[i + 1] - K[i + 3]

= P[i] + P[i + 1] - P[i + 3],

since K[i + 3] = K[i + 1] + K[i] by definition.

By applying this transformation, we eliminate all reference to the key, and are left with a simple keyless transformation on the plaintext.

If P[i + 1] = P[i + 3], then C'[i] = P[i]. So if we find three successive identical values of C'[i], we can be fairly confident that these reflect successive identical values of P[i], and can use this information to deduce the key.

All word-processing files created by Wordstar have bytes of 1A appended to them to pad them to a multiple of 128 bytes. This means that assuming the last three words of the file to be 1A1A has a 95% probability of success.

We have written a program that looks at the last four words of a file, checking to see if they can be all 1A1A; failing that, it looks through the file for successive identical values in the way described. On the test disk, this program failed on 9 out of 141 files: 6 documents under 800 bytes, one 1600-byte document, and two Basic programs. In each case, the program takes less than a second to deduce the key and start decrypting the file.

An interactive version of this program would ask the user for a word or phrase to search for, ("because", "from the", etc.): this would increase the success rate to nearly 100%, irrespective of the software package that produced the file.

Padlock "*looks like becoming the standard in PC data protection and security systems*".

PS3

PS3 (PS Level III) is one of a range of security products, including the program protectors PS1 and PS2. *"The PS Protection System provides total security in the microcomputer environment... and can be regarded as a fifth generation product*". It uses "*... a variant of the United States Government's Data Encryption Standard*"

PS3 maintains a system of passwords and authorisations which can be of Byzantine complexity. It keeps a directory of all the files it protects, and when a protected file is opened, it checks the user's password, looks up the encryption key, and uses it to encode or decode as necessary. There are various technical drawbacks and performance penalties in the PS3 system, which relies heavily on intercepting and then emulating operating system calls, but here we are concentrating on the cryptographic strength of the system, without which no protection scheme, however complex, can be effective.

Files encrypted with PS3 use an inherent key *only*. Keys are recorded in the directory along with the passwords and access authorisations for each file.

The algorithm

PS3 uses three 8-bit keys: K0, K1[0], and K1[1]. Plaintext is treated as a sequence of bytes, P[i], with P[0] as the first byte. To encrypt a plaintext byte P[i] into a ciphertext byte C[i]:

1. x := P[i] XOR K1[i mod 2]
2. Rotate x right by 2 bits.
3. If x has even parity (i.e. an even number of 1-bits in its binary representation) set the top bit of x to 1; otherwise set the top bit of x to 0: so that $01 stays as $01, but $41 becomes $C1.
4. C[i] := x XOR ((K0 + i) mod 256)

By defining K1 differently, one could postpone step 1 and make it part of step 4; but the way that keys are stored in the directory argues conclusively for the order shown.

The PS3 algorithm is outstandingly simple to deduce by the usual method of encrypting a file consisting entirely of zero bytes: the pattern is obvious to the eye, and only a few test encryptions are needed to disentangle steps 2 and 3.

The attack

If we split alternate bytes of P and C into separate streams (C0[i] = C[2 * i] and C1[i] = C[2*i+ 1] etc.), and look at each stream separately, then steps 1 to 3 become simple position-independent substitutions. We can then start finding K0 without worrying about the values of K1.

The entropy of a message can be considered as a measure of its randomness - or equivalently, of how many bits are required for a minimal encoding of the message. The text of this paper has an entropy of 4.79 bits per byte, or 5.30 bits if Wordstar's top bit flags are included, and even executable program files have entropies of under 7.5 bits per byte.

We calculate the entropy of a message as the sum (from i = 0 to 255) of -f(i) * log(f(i)), where f(i) is the relative frequency of the byte value i in the message (so if half the characters in the message are spaces (ASCII code = $20), then f($20) = 0.5), and logarithms are to base 2.

Simple substitutions do not affect the entropy of a message; thus steps 1 to 3 result in a stream which has the same entropy as the original message. Step 4 does alter the entropy of the message because it applies 128 different substitutions to different bytes within C0, and similarly for C1. Moreover, in scrambling the message it will tend to increase the entropy, since it is adding disorder to the message; the probability of a decrease in entropy is negligible.

To find K0, try decoding the file with every possible value of K0 (there are only 256 of them), and calculate the entropy of each such decoding. Assume that the value of K0 yielding the lowest entropy is the correct one. (It is highly improbable that a wrong key should produce a decoding less random - more message-like - than the real message).

Using both C0 and C1 (separately) will give a value of K0 which is unique except for its top bit. This ambiguity is a consequence of the fact that adding $80 to a number (modulo $100) is the same as exclusive-ORing it with $80; and it does not matter, because a "wrong" choice of K0 will automatically correct itself by changing the top bits of the K1 to correspond.

Once K0 has been deduced, steps 4, 3, and 2 can be reversed, and we only have to find the values of K1[0] and K1[1] to undo step 1 and break the whole cipher.

Two adjacent characters of known plaintext would be enough to deduce K1[0] and K1[1]. We can already deduce, from the ciphertext alone, the value of x calculated in any particular instance of step 1, and so (given P[133], say) we have x = P[133] XOR K1[1], where x is known; whence K1[1] = P[133] XOR x.

In a purely automatic system, however, the best thing is to decide what the distribution of the plaintext characters ought to look like. In our test program we have simply assumed that the space (code $20) is the commonest character, as is usually the case in text files.

A more sophisticated approach would be to assume that the distributions of P0 and P1 are the same, and choose a value of K1[0] XOR K1[1] which maximises their coefficient of correlation: this

would leave us with only one byte to guess! But there does not seem to be any need to go this far in practice.

"*There is NO GENERIC breaking mechanism for PS disks*". Our test program will decrypt practically any file encrypted with PS3, without knowing the serial number of the encrypting copy of PS3 or looking at the directory of keys. On our test disk, the program failed on 13 out of 141 files, all of them under 900 bytes long.

All information about protected files is held on a special directory file. Reading this file would give a complete list of filenames, passwords, and keys, making it a trivial task to read any file. This file is encrypted "*with a different encryption algorithm from the data file protection mechanism.*" The algorithm is actually the same: only the key is different, and our program decodes the directory file without difficulty.

"*Of course, any code can be unscrambled eventually but it requires vast amounts of time and equipment costing hundreds of thousands of pounds.*" But PS3 needs a program less than a kilobyte long, and the key is found in under 30 seconds!

CRYPT

Crypt is an attempt at a transparent encryption system. It uses inherent keys only (no user keys, no file keys), and intercepts the loading and saving of files by Lotus 1-2-3 and Symphony, encrypting as files are saved and decrypting as they are loaded. The program is supplied with warnings against using many normal DOS facilities while Crypt is running; additionally, the mechanism used for detecting which files are encrypted seems prone to error. No sensible person would therefore use the program as it stands; but if these problems were corrected, would it be cryptographically secure?

The algorithm

Crypt uses two 128-byte keys, K and R. Plaintext P is encrypted into ciphertext C by:

1. n := P[i] rotated left by R[i mod 128] bits.
2. C[i] := n XOR K[i mod 128].

K and R probably have some sort of internal structure, but we shall not assume any. This means that we have to deduce 128 separate byte values for each key - effectively solving 128 ciphers simultaneously. One needs quite a quantity of text to get good enough statistics to crack the cipher, but that is made less of a problem by the fact that everything is encrypted with the same key.

The attack

We treat Crypt as 128 separate ciphers. The first acts on the Ist, 129th, 257th,... bytes of each file; the second on the 2nd, 130th, 258th,... and so on. The next stages of the attack are performed 128 times, once for each of these ciphers.

Note the frequency with which each of the 256 possible ciphertext byte values occurs. Next, find the bits whose frequency is skewed most (i.e. far more 1s than 0s or vice versa) - for instance, if there are far more (or far fewer) odd ciphertext byte values than even ones, then the least significant bit is considered to be heavily skewed.

Almost all ASCII text has its top bit ($80) heavily skewed. Most ASCII text also has its top-but-two bit ($20) skewed, because upper-case and lower-case letters differ in this bit, and it is rare for a sample of text to contain equal proportions of upper-case and lower-case letters. Thus the most skewed bits in the ciphertext are likely to reflect the most significant bits of the original text, and this allows us to guess R. Having guessed R, guess K by assuming that the space ($20) is the commonest character in the file.

If there were enough text, these values of R and K would be correct. All files use the same keys, and so files may be lumped together when accumulating frequencies; but it may still be necessary to work with fairly small samples of text, so we have implemented the decryption process in three programs. The first uses the method shown above to deduce a tentative key schedule for all 128 sub-ciphers. The second program, which is interactive, allows the user to inspect the text of a decrypted file, and, by editing obviously wrong characters, to alter the key table and instantly see on the screen all the resultant changes in the decrypted file. Since people are far better than computers at recognising patterns, it only takes a fairly short session to correct any inaccuracies in the deduced keys. The third program takes the modified key table and decrypts files with it.

Because of the human element, it is impossible to give a firm success rate for the attack on Crypt; but only 10 kilobytes or so of text are usually needed to effect a successful decryption, and the use of the same key for all encrypted files is a major advantage.

N-CODE

The designers of N-Code claim to be well aware of the dangers of simple substitution ciphers under known-plaintext attack, however the key streams are constructed, and they emphasise this in their tutorial program: "*With both an original and encoded version of the same material, a clever thief can ... compare them and examine the pattern of an ordinary substitution code... other encoded documents will be compromised. With N-CODE, the characters are re-arranged as well as [substituted]. The scrambling pattern is different for every copy of N-CODE, every user-selected key, and even every individual file. A thief will have to try TRILLIONS of combinations...!*"

N-Code uses a substitution-transposition cipher in which the transpositions are dependent on the content of the text being encrypted. It uses both an inherent and a user-selected key, neither of which is stored in the file.

The algorithm

The N-Code algorithm is more elaborate than the others we have described because of the transposition step; but the basic structure can be fairly easily deduced from a few trial encryptions.

Keys are:

N, a number between 100 and 190;

A, an array of N 16-byte vectors;

R, an array of 85 permutations on 16 elements;

B, an array of 23 numbers between 0 and 84.

N-Code acts on 16-byte blocks of plaintext. We shall number the blocks from 0, and the bytes within the blocks from 0: so the 3rd block of the file is P[2], and the 4th byte of the 1st block is P[0, 3].

To encrypt P[j] into C[j]:

1. For i := 0 to 15

   n[i] := (P[j, i] + A[j mod N, i]) mod 256.
   (substitution step)

2. k := (n[0] + n[1] + ... + n[15]) mod 85

   j1 := (j mod 32) mod 23
   (select a transposition)

3. C[j] := n permuted by R[k + B[j1] mod 85].
   (transposition step)

Note that step 3 does not alter the sum of the elements of a block, so a file can be decrypted by performing step 2, then inverting steps 3 and 1 in that order.

The attack - N

Since the makers of N-Code specifically claim that it is proof against a known-plaintext attack, it seems reasonable to try one. N-Code is the only cryptosystem so far considered that does not yield instantly and trivially to a known-plaintext attack.

The first thing to find is the value of N.

Write $DP^N[j, i]$ for P[j + N, i] - P[j, i], and $DC^N[j, k, m]$ for C[j +N, k]- C[j, m].

If we have the correct value of N, then for all i and j,

$$DP^N[j, i] = n[j + N, i] - n[j, i]$$

Since C is a permutation of n, it follows that for any j and i there exist k and m such that

$$DP^N[j, i] = DC^N[j, k, m].$$

There are always gaps in the 256 possible values of $DC^N[j, k, m]$, and so if we choose the wrong value of N, we will quite soon find values of j and i such that $DP^N[j, i]$ hits one of those gaps.

In practice, no more than 4 blocks need to be examined to narrow the search down to the correct value of N.

The attack - A

The next thing to calculate is the values of A.

We look again at the differences DP and DC (we omit the index N now that the correct value of N has been deduced).

For a given value of j and i, the equation DP[j, i] = DC[j, k, m] may have several solutions (k, m) or just one. If there is a unique solution, this gives a value of A[j, i], since

A[j, i] = C[j, m] - P[j, i]

(This also fixes two points in the transpositions, but we ignore this information for the moment).

Typically, several values of i yield unique solutions (k, m); moreover, no value of k or m can be re-used, so once some values of i have been matched, other values of i which originally matched several values of (k, m) will be found to match only one that does not overlap with a previous solution. In many cases, all 16 bytes of A[j] can be deduced from consideration of DP[j], without the need to look at A[j + N], A[j + 2 * N], etc.; in some cases, two or three bytes in A[j] remain undefined.

Our first attempt at a known-plaintext attack scanned through the file until the whole of A was deduced. It then became apparent that A was constructed as follows:

1. There is a master table AM, 78 bytes long.
2. Each value of A[j] is made up of AM[a1], AM[a1 + 1], ..., AM[a1 + a2] followed by the first 15 – a2 values of AM. The values of a1 and a2 depend on j in no obvious way.

It follows that once AM has been deduced, the gaps in any value of A can be filled by referring to its construction from AM.

This means that the construction of table A can be done quite efficiently: first deduce enough values of A (ignoring difficult cases) to construct AM; then deduce *all* the values of A by reference to AM where necessary.

The attack - R and B

Finally, we need to deduce R and B.

Given A and N, it is a simple matter to deduce the particular permutation that has been used in encrypting each C\j\.

- Looking at the permutations applied to two blocks with identical values of j1, the same value of B will be used, so the only cause of difference in permutation is differing values of k. Thus we can establish the distance apart of the two permutations in the table R.
- If the same permutation occurs for two blocks with different values of j1, then we can use the difference in the values of k to deduce the difference in the corresponding values of B[j1].

Scanning through the file, a combination of these two factors will soon allow the whole of R and B to be deduced.

Once the key table has been constructed, it can be used to decrypt all files encrypted with the same key.

The description of the attack has been long, because the encryption algorithm itself is quite elaborate; but there is nothing complex in it, and the program to deduce the key from known plaintext takes only a few minutes to run.

ULTRALOCK

We wrote Ultralock, so we do not propose to describe it in detail. However, here is a summary of its features for purposes of comparison:

- No inherent keys.
- Keys are 16 bytes long, generated as a digest from 8- to 60- character user-entered keys.
- Different keys can be assigned to different files or groups of files.
- Operation is completely transparent: data are encrypted just before being written to the disk by an application or by DOS; and decrypted just after being read from the disk.
- Proprietary (but DES-inspired) encryption algorithm operates on 128-byte blocks, with complete diffusion within a block.
- Key management facilities.

SUMMARY

All the packages we have looked at suffer from a lack of diffusion: a small change in the plaintext causes only a small change in the ciphertext. Superkey and Crypt operate on each byte separately; Padlock and PS3 operate on each word separately; and N-Code operates on each byte separately and then disguises this by permutations over 16 bytes. These properties make it easier to break a cipher when only local properties of the plaintext are known.

As a consequence of this lack of diffusion, all these packages are vulnerable to known-plaintext attacks; some with as few as 6 bytes of plaintext needed. Moreover, all the packages except N-Code can be broken quite effectively without any known plaintext at all.

We have not described our impression of the usability of these packages, many of which require such elaborate procedures to operate them that in practice people will not use them at all.

All users can form their own opinions on usability. But most users cannot form opinions on the cryptographic security of the packages they are considering. Encryption systems are unique among software packages in that the user cannot readily tell if a package is doing its job properly. If a compiler or a word-processor has errors, they are usually obvious; but a weak cryptosystem looks just like a strong one in operation.

The claims we have quoted throughout this paper are the only cryptographic information that users have when they are evaluating the security of systems; and, as we have seen, these claims are spurious. Security program suppliers must behave more responsibly, and realise the importance of supplying genuinely secure systems.

PRODUCTS MENTIONED

SuperKey from Borland International, 4585 Scotts Valley Drive, Scotts Valley CA 95066 USA.

Padlock from Sovereign Software Ltd, 45 Southfield Road, Southfields, London SW18 5EZ, England.

PS3 from Stralfors Data Products, 11 Techno Trading Estate, Swindon SN2 6HB, England.

Crypt from BCS, Frankfurt, West Germany.

N-Code from K+L Software, 11425 Oak Leaf Drive, Silver Spring MD 20901 USA.

Ultralock from Business Simulations Ltd, Scriventon House, Speldhurst, Kent TN3 0TU, England

BIOGRAPHICAL SKETCH

Martin Kochanski holds a BA in Mathematics and Philosophy from Balliol College, Oxford. He works for Business Simulations Limited, a software house supplying database and security packages for personal computers, and has designed the world's first commercially available chip for RSA encryption. He does not intend to start a validation service for rival encryption and security products.