A Chosen-Plaintext Attack on the Microsoft BASIC Protection

R. van den Assem
Delft, The Netherlands

and

W.J. van Elk
Delft, The Netherlands

The Microsoft BASIC (MBASIC) interpreter provides a command which protects the program currently in memory by saving it, on disk, in an encrypted format. A user can RUN such a protected program, but cannot access the source program. A chosen-plaintext attack was used to break the encoding; the encryption method could be derived easily from the enciphering of carefully chosen plaintext programs. As a result, a pair of MBASIC programs able to decrypt a protected program for any interpreter was developed. Further, it is shown that a secure system can never be realized, whichever encryption method is used.

Keywords: Microsoft BASIC interpreter, software protection, cryptology, chosen-plaintext attack, plaintext, ciphertext, encryption, decryption, security

René van den Assem is a student of Delft University of Technology (Department of Mathematics and Informatics), majoring in computer science.

Willem-Jan van Elk is a student of Delft University of Technology (Department of Mathematics and Informatics), majoring in computer science. To obtain his MSc degree he is currently making researches into parallel computation and its use in combinatorial optimization.

North-Holland
Computers & Security 5 (1986) 36–45

1. Introduction

The Microsoft BASIC interpreter provides a command, "SAVE ⟨filename⟩,P", which saves the program currently in memory in an "encoded binary format" [1]. After reloading an encrypted program the user can RUN the program, but cannot use certain commands, such as LIST, EDIT, PEEK or POKE, to access the source program in a regular way. Because of our interest in cryptology we accepted the challenge to find out which encoding was used and how it could be broken. (This article describes how we broke the code. [2] is the complete research-report.) The goal we set ourselves included finding out the exact encryption algorithm used by pure cryptologic means and developing a decryption computer program. As the codebreaker in this case can choose any plaintext he likes and can examine the corresponding ciphertext, a *chosen-plaintext attack* [3] was the obvious approach.
All examples in this article are for the Microsoft BASIC interpreter rev. 5.03, CP/M version, on an Exidy Sorcerer microcomputer, though this does not restrict the generality of the attack (see Section 11). Throughout this article hexadecimal representation will be used for memory contents.

2. The SAVE command

A program can be saved in three different formats [1]:

1. ASCII format ("SAVE ⟨filename⟩,A")
   The program is saved as a sequence of characters in ASCII, in the way it would appear on screen after a LIST command.

2. Compressed binary format ("SAVE ⟨filename⟩")
   This format is a byte by byte copy of the memory representation of the program. In this format BASIC command names are represented by one byte, a so-called token. Line numbers

are represented by two bytes. The representation of each line starts with a two- or three-byte link to the start of the next line. The following example shows two-byte links. (The hex dumps of the byte sequences in this example are made by a computer program to be described in Section 3.)

The program

10 REM demo
20 END

will be stored internally as in Table 1. The "SAVE ⟨filename⟩" command saves this program as the byte sequence:

FF F8 5F 0A 00 8F 20 64 65 6D
6F 00 FE 5F 14 00 81 00 00 00

3. Encoded binary format ("SAVE ⟨filename⟩,P") The program is saved in an encrypted (protected) form. The example program given above is saved as the byte sequence:

FE 30 1F 2B C1 61 F7 CB 05 39
3C AA 0F 6E 21 10 D8 B2 B2 0B

3. First Investigation

Our first concern was to find out which was the plaintext's format to be enciphered. We assumed that this could be either the ASCII form or the tokenized form. (So we also assumed that the interpreter did not generate an intermediate format to serve as the plaintext.) For testing this assumption we took a large program which we saved in the three different formats, in order to compare the times required for the respective SAVE processes and the sizes of the resulting files on disk. The SAVE times and file-sizes are given in Table 2.

Table 1

Address	Byte	Comment
5FED	F8	address of start next line
5FEE	5F	
5FEF	0A	line number
5FFO	00	
5FF1	8F	token for REM
5FF2	20	ASCII code for space
5FF3	64	ASCII code for d
5FF4	65	ASCII code for e
5FF5	6D	ASCII code for m
5FF6	6F	ASCII code for o
5FF7	00	end of line
5FF8	FE	address of start next line
5FF9	5F	
5FFA	14	line number
5FFB	00	
5FFC	81	token for END
5FFD	00	end of line
5FFE	00	end of program
5FFF	00	

Table 2.

SAVE format	Time	File-size
ASCII	23.5 s	138 recs
tokenized	17.7 s	136 recs
protected	21.6 s	136 recs

Since the file-sizes of the three different formats were nearly identical, the differences in SAVE times must have been caused almost solely by CPU operations needed to create the different formats. As we knew the tokenized format to be the format used for storing the program internally, we expected a minimum of such operations (and therefore a minimum SAVE time) to be needed for this format, as proved to be the case. Given the fact that the SAVE time for the ASCII format was significantly larger than the SAVE time for the protected format we were able to rule out the possibility of the ASCII format functioning as plaintext for the enciphering process. We therefore concluded that the most likely plaintext was the tokenized form.

Given this assumption, one can see from Table 2 that the encryption of 136 records, each 128 bytes long, takes 3.9 seconds, so that the encryption of a single byte takes about 224 μs (448 clock cycles). The Z80 has no instruction for multiplication but a typical 8-bit binary multiplication in machine language takes about 400 clock cycles [5]. Thus, a cipher algorithm using a multiplication is not very likely. This observation tends to rule out, for example, the possibility of a stream cipher [3] based on a linear congruential sequence [4].

In order to investigate the encoding we wrote a program to print out the ASCII form (listing) of a program as well as hex dumps of the tokenized form (plaintext) and the protected form (ciphertext). Using this program we performed some simple tests to see what elementary properties the encoding has, such as repetitivity and context-

sensitivity. From these tests we concluded the following:

1. The first byte of a program-file indicates whether the program is encrypted or not. For an encrypted program the first byte is fixed at FE while for an unencrypted program this is FF.
2. The encoding is purely local. Changing a single byte in the plaintext at position i caused a change in the ciphertext at position i only. (This fact excluded the possibility of a stream cipher.)
3. The encoding is position-dependent, which means that the encoding for a given byte depends on its position in the plaintext.
4. The encoding appeared to be repetitive with a period of 143. Further investigations for this property were carried out (see Section 4).

With properties 2 and 3 in mind we use the notation $T_i(X)$ for the encryption of byte X at position i. The low byte (that is: the first byte) of the link of the first line is defined to be position 1. (Note that this is the second byte of the tokenized file.)

4. Verification of the Repetitivity

Our next step was to verify that the encoding was repetitive with a period of 143. We therefore created a program, as large as possible, with lines of 143 bytes (in tokenized form) wherein the bytes representing the line numbers and the links to the start of the next line were different from line to line, but the other 139 bytes were identical between lines. This program was saved in protected form, and the resulting code-file was tested for repetitivity. Of course the bytes corresponding to line numbers and links were disregarded. As expected, the code-file appeared to be repetitive with a period of 143.

Although this did not at all amount to a mathematical proof, we accepted the repetitivity of the encoding as a fact. That is,

$$\forall i,\ X\colon T_i(X) = T_{i+143}(X) \tag{1}$$

(The processes of generating the repetitive program described above and checking the repetitivity of the corresponding code-file were carried out by computer programs.)

5. Creating the Tables

Equation (1) implies the possibility of creating 143 tables, for T_1 through T_{143}, which describe the encoding completely. Because we had, up to this point, no idea of the nature of the encoding we indeed created a complete set of tables. We therefore developed a program that POKEd systematically the number 0 (the 00-byte) through 255 (the FF-byte) into a position in memory corresponding with the low byte of one of the line numbers of the program. After such a POKE instruction the program SAVEd itself (!) in protected form and OPENed the code-file as data to observe the

Table 3.

H/L	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	CF	44	45	42	43	48	49	46	47	3C	3D	3A	3B	40	41	3E
1	3F	34	35	32	33	38	39	36	37	2C	2D	2A	2B	30	31	2E
2	2F	24	25	22	23	28	29	26	27	1C	1D	1A	1B	20	21	1E
3	1F	14	15	12	13	18	19	16	17	0C	0D	0A	0B	10	11	0E
4	0F	84	85	82	83	88	89	86	87	7C	7D	7A	7B	80	81	7E
5	7F	74	75	72	73	78	79	76	77	6C	6D	6A	6B	70	71	6E
6	6F	64	65	62	63	68	69	66	67	5C	5D	5A	5B	60	61	5E
7	5F	54	55	52	53	58	59	56	57	4C	4D	4A	4B	50	51	4E
8	4F	C4	C5	C2	C3	C8	C9	C6	C7	BC	BD	BA	BB	C0	C1	BE
9	BF	B4	B5	B2	B3	B8	B9	B6	B7	AC	AD	AA	AB	B0	B1	AE
A	AF	A4	A5	A2	A3	A8	A9	A6	A7	9C	9D	9A	9B	A0	A1	9E
B	9F	94	95	92	93	98	99	96	97	8C	8D	8A	8B	90	91	8E
C	8F	04	05	02	03	08	09	06	07	FC	FD	FA	FB	00	01	FE
D	FF	F4	F5	F2	F3	F8	F9	F6	F7	EC	ED	EA	EB	F0	F1	EE
E	EF	E4	E5	E2	E3	E8	E9	E6	E7	DC	DD	DA	DB	E0	E1	DE
F	DF	D4	D5	D2	D3	D8	D9	D6	D7	CC	CD	CA	CB	D0	D1	CE

code-byte. In this way the complete code-tables for T_{10} through T_{165} were acquired. The observation that

$T_{10} = T_{153}$

$T_{11} = T_{154}$

etc.

confirmed the repetitivity of the encoding once more. Table 3 is the table of T_{13}, which will be used as an example through the remainder of this article. In this table H and L are the high nibble and low nibble respectively of the byte to be encoded. (A nibble is a half-byte, a group of four bits.) As an example: at position 13 the byte 5F is enciphered as 6E.

6. Investigation of the Tables

One of the most striking properties of T_{13} (Table 3), and of all other tables T_i, is the fact that the low nibble encryption (LNE) is independent of the value of the high nibble. The encryption $t_{13}(x)$ of low nibble x for position 13 is given in Table 4. (We will use lower case characters for nibble variables, operators and functions to distinguish them from their byte analogues.)

7. The Low Nibble Encryption

All LNE tables showed a similar structure, in the way they were built up from blocks. (A block is a set of consecutive hexadecimal numbers. In this section 0 is assumed to be the successor of F. Therefore, the table of t_i is cyclic in x.) The table of t_{13}, as do all other tables t_i, shows 8 blocks of size 2 (4-5, 2-3, 8-9, and so on), grouped in 4 blocks of size 4 (4-3, 8-7, C-B and 0-F), in their turn grouped in 2 blocks of size 8 (4-7 and C-F) and, trivially, in one block of size 16 (4-F). For Table 4 the partition in blocks is depicted by the brackets. (Note that the partitions are not always uniquely determined; in t_{13} we could also have chosen 8-B and 0-3 as the blocks of size 8.)

For a given table and partition the order of the two half-blocks in building up a block depends only on the blocksize. For example in t_{13}, all blocks of size 2 are in natural order while the blocks of size 4 all start with the higher half-block and end with the lower half-block (such as 4-5/2-3). Now remarkably the same structure appears also in the table for a simple bitwise exclusive-or operation. See as an example Table 5 for

$$t(x) = x \text{ xor } 2 \tag{2}$$

The LNE tables differed however in two aspects from a pure xor-encoding like $t(x)$:

1. In a pure xor-encoding the 16 numbers are consistently partitioned into blocks (for example the numbers in the blocks of size 4 are always 0-3, 4-7, 8-B and C-F) which is not the case for the LNEs. This suggests that an addition is performed *after* the xor operation.
2. The blocks in a pure xor-encoding always start in the same position (as can be seen in Table 5, $t(0)$ is the start of a block for each blocksize) which is not the case for the LNEs. This displacement suggests that an addition is performed *before* the xor operation as well.

Our hypothesis for the LNE for position i therefore became:

$$t_i(x) = ((x + a_i) \text{ xor } q_i) + b_i \pmod{16} \tag{3}$$

where (a_i, b_i, q_i) is a position-dependent triple of nibbles. We next wrote a program that for a number of consecutive LNE tables generated all triples (a_i, b_i, q_i) that would specify, by (3), the same encryption function.

For each of these tables 8, 16 or 32 such triples were found. The 8 triples (a, b, q) found for t_{13} are given in Table 6.

We next tried to select one triple for each LNE such that a pattern would emerge in triples for consecutive positions. For a_i and b_i the following

Table 4.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$t_{13}(x)$	F	4	5	2	3	8	9	6	7	C	D	A	B	0	1	E

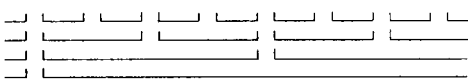

Table 5.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
t(x)	2	3	0	1	6	7	4	5	A	B	8	9	E	F	C	D

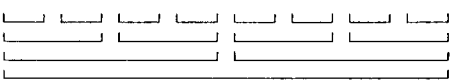

Table 6.

a	b	q
3	E	2
3	6	A
7	A	2
7	2	A
B	6	2
B	E	A
F	2	2
F	A	A

pattern was easily recognized:

$$\begin{cases} a_1 = 3 \\ a_{i+1} = \begin{cases} a_i + 1 & \text{if } a_i \neq \text{F} \\ 3 & \text{if } a_i = \text{F} \end{cases} \end{cases}$$
$$\begin{cases} b_1 = \text{B} \\ b_{i+1} = \begin{cases} b_i - 1 & \text{if } b_i \neq 1 \\ \text{B} & \text{if } b_i = 1 \end{cases} \end{cases} \quad (4)$$

So, the triple we chose for t_{13} was (F, A, A). For q_i no pattern was recognized. It is easy to see that, since a_i varies from 3 up to F and b_i varies from B down to 1, their periods are 13 and 11 respectively. It seemed logical to assume that q_i was uniquely determined by a_i and b_i, because the period p of the encoding could then be computed to be

$$p = \text{LCM}(13, 11) = 13 \cdot 11 = 143$$

We found that three other recurrence relations for a_i and b_i were possible in which a_i and b_i additively differed from the ones described by (4) by multiples of 4. We have chosen equations (4) because then both a_i and b_i make the jump when they are to become zero, which for an assembly language programmer is slightly easier to implement.

We next wrote a program which verified that (4) holds for all 143 LNEs and simultaneously built up a table for the corresponding q_i's by extracting q_i from table t_i. Indeed for every table a triple (a_i, b_i, q_i) with a_i and b_i satisfying (4) was shown to exist, so equation (3) was no longer hypothetical. However, the sequence of q_i's still did not display any obvious regularity. Our next concern then was the structure of the complete encryption tables.

8. The Complete Encryption

Looking through the complete encryption tables we observed the following:

1. In each encryption table at most 4 different high nibble encryption (HNE) tables were used (in our example Table 3, T_{13} has three different HNE tables, with first nibbles C, 4 and 3 respectively). The structure of these HNE tables was identical to that of the LNE tables described in the previous section.
2. The structure of the complete encryption tables showed remarkable similarities to the structure of the LNE tables. Especially the partition in blocks could be carried out further (blocks of size 16, 32 up to 256) and this time not only for nibbles but for entire bytes.

These observations suggested that the complete encryption tables might very well have been constructed in the same way as the LNE tables. We therefore formulated as a hypothesis for the complete encryption scheme:

$$T_i(X) = ((X + A_i)\text{XOR}\, Q_i) + B_i \pmod{256} \quad (5)$$

where (A_i, B_i, Q_i) is a position-dependent triple of bytes, and XOR is the bitwise exclusive-or operation on bytes. In order to verify this hypothesis we wrote another computer program to generate triples (A_i, B_i, Q_i) for a number of encryption tables, assuming that the low nibbles of A_i, B_i and Q_i were the a_i, b_i and q_i respectively as found in our earlier work.

This approach proved to be successful and we found that choosing F and 0 as high nibbles for A_i and B_i respectively was all we needed to obtain an acceptable behaviour of A_i and B_i. Table 7 shows the possible triples (A, B, Q) for position 13 with the assumptions regarding the low nibbles of A, B and Q given above.

We found the high nibble of Q_i to vary just as

Table 7.

A	B	Q
3F	CA	3A
3F	4A	BA
7F	8A	3A
7F	0A	BA
BF	4A	3A
BF	CA	BA
FF	OA	3A
FF	8A	BA

mysteriously as did q_i. This suggested that Q_i was some fixed function of A_i and B_i,

$$Q_i = Q(A_i, B_i) \tag{6}$$

probably in the form of a key. Assuming Q to be a 143 byte key we decided to extract that key out of the encryption tables, again by means of a small computer program. The value of Q_i was computed from the encryption of the 00-byte in the i-th table, using

$$Q_i = (T_i(00) - B_i)\text{XOR}\, A_i \pmod{256} \tag{7}$$

which follows from (5). Having extracted the Q-key we reached our ultimate goal by writing a computer program, in Microsoft BASIC of course, able to decrypt an encrypted program, using the inverse of (5):

$$T_i^{-1}(X) = ((X - B_i)\text{XOR}\, Q_i) - A_i \pmod{256} \tag{8}$$

9. Selective Disassembly

After our cryptologic adventures we wanted to disassemble parts of the MBASIC interpreter, mainly as a last form of verification that the encryption was as assumed. We first noted that in the computation of $T_i(X)$, the additions could very well be implemented as subtractions, for example the addition of A_i because the behaviour of A'_i (256 minus A_i) would then be identical to that of B_i (apart from the period of course). We then wrote a program that PEEKed in the machine code of the interpreter looking for an ADD, XOR, and ADD instruction as consecutive instructions. Such a combination of instructions, however, was not found.

We modified this program so that it would search for ADD-XOR, SUB-XOR, XOR-ADD and XOR-SUB combinations. These combinations occurred 2, 2, 4 and 4 times respectively. Four of these pairs, two SUB-XOR and two XOR-ADD combinations, were lying in a very small address interval. When we disassembled the corresponding part of the interpreter we found two routines that immediately could be identified as the encrypting and decrypting routines respectively.

These routines indeed showed a subtraction of A'_i, instead of the addition of A_i described by (5), and, more or less surprisingly, some regularity in Q_i after all, namely

$$Q_i = Q(A'_i, B_i) = Q_D(A'_i)\text{XOR}\, Q_B(B_i) \tag{9}$$

That is, the Q-key which we found to be 143 bytes long was in fact built up from two smaller keys, a 13 byte key Q_D and an 11 byte key Q_B, indexed by A'_i and B_i respectively, as specified in Tables 8 and 9. (The subscripts D and B denote the length of the key in hex.) Both keys are part of the machine code of the interpreter!

10. Summary

The complete encryption scheme is as follows:

$$T_i(X) = ((X - A'_i)\text{XOR}\, Q_i) + B_i \pmod{256}$$

$$\begin{cases} A'_1 = D \\ A'_{i+1} = \begin{cases} A'_i - 1 & \text{if } A'_i \neq 1 \\ D & \text{if } A'_i = 1 \end{cases} \end{cases}$$

$$\begin{cases} B_1 = B \\ B_{i+1} = \begin{cases} B_i - 1 & \text{if } B_i \neq 1 \\ B & \text{if } B_i = 1 \end{cases} \end{cases} \tag{10}$$

$$Q_i = Q_D(A'_i)\ \text{XOR}\ Q_B(B_i)$$

where Q_D and Q_B are as specified in Tables 8 and 9 respectively.

Table 8.

A'_i	1	2	3	4	5	6	7	8	9	A	B	C	D
$Q_D(A'_i)$	FB	D7	1E	86	65	26	99	87	58	34	23	87	E1

Table 9.

B_i	1	2	3	4	5	6	7	8	9	A	B
$Q_B(B_i)$	4A	D7	3B	78	02	6E	84	7B	FE	C1	2F

11. Extension to Other Machines and Interpreters

We examined a number of 8080- and Z80-based machines with MBASIC interpreters of widely varying version numbers. In each of these cases we found our algorithm to be immediately applicable, that is, even the Q-keys were the same. Some investigations on machines with processors other than the 8080 or Z80 showed that only the Q-keys were different, which was no surprise since the keys are machine code. We finally wrote a universal pair of MBASIC programs; the first able to extract the 143 byte Q-key for the specific interpreter under which the program is executed, and the second able to encrypt and decrypt (using the Q-key found by the first program). These programs are listed in Appendix A.

12. Conclusions and Discussion

1. The designers of the MBASIC interpreter implemented a very simple encoding that, as shown by the research described above, was easy to break. With little effort the encoding could have been made more difficult to break. We suggest:

a. A'_i and B_i should have been given much longer periods, so that creating all the tables would no longer be feasible.

b. A'_i and B_i should have been given different decrements, other than 1, so that a pattern in the behaviour of A'_i and B_i would be more difficult to recognize.

c. Q_i should have been given a behaviour independent of those of A'_i and B_i, so that the period of the encoding would be (much) larger than the address space of the 8080 or Z80.

In all these cases, however, the nature of the encoding can be discovered by examining a few encryption tables, in which case the encoding is, practically speaking, broken. Therefore, suggestions like the ones given above are no real improvements.

2. The designers may have thought that the encoding had to be local (see Section 3). With a local encoding, a read error for a certain byte would destroy only a single byte of the program, instead of the rest of the program as in the case of the stream cipher. There is, however, for the user no regular way to correct an erroneous byte. So destruction of a great number of bytes is no worse than destruction of a single byte. (It may even be considered as better, because the non-correctness of the program is easier for the user to detect.)
We therefore suggest the use of an encoding which is not local, for example a stream cipher. It is obvious that such an enciphering does not need to take much CPU time, while it makes the job of the cryptanalysist much more difficult. Given the facts that the deciphering and enciphering probably have to be fast and that the 8080 (Z80) has only a very limited instruction set (for example without a multiplication instruction) we believe that the encoding has to be simple and hence that a thorough and systematical investigation will eventually also break this encoding.

3. Every MBASIC user has access to not only the encrypting but also the decrypting routines, because these are parts of the interpreter. So the burglar can break the encoding, though not very elegantly, by disassembling the interpreter and following the paths of the "SAVE ⟨filename⟩,P" and "LOAD" commands. Even if the designers had been able to implement an encryption that can resist a chosen-plaintext attack (for example a public-key cryptosystem [3]) they would not have made a major contribution to the security of the protection, because security is broken by making the machine code of the interpreter accessible.

Acknowledgement

The authors wish to express appreciation to prof. dr. I.S. Herschberg and drs. J.W.J. Heijnsdijk for their many helpful suggestions and comments as well as their critical reading of this article.
We also like to thank our colleague-students J. Adriaanse and M.J. Drużdżel who did some preparatory work on the Microsoft BASIC encryption method and who brought the problem to our attention.

References

[1] Microsoft BASIC Reference Manual (doc. no. 8101-530-08). Microsoft, 1979.

[2] R. van den Assem en W.J. van Elk: Computerveiligheid Taakverslag (in Dutch). Department of Mathematics and Informatics, Delft University of Technology, August 1984.

[3] D.E.R. Denning: Cryptography and Data Security. Addison-Wesley, Reading, Massachusetts, 1982.

[4] D.E. Knuth: The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 2nd edition, Addison-Wesley, Reading, Massachusetts, 1981.

[5] L.A. Leventhal: Z80 Assembly Language Programming. Osborne/McGraw-Hill, Berkeley, California, 1979.

Appendix: The Final Programs

In this appendix the two MBASIC programs, mentioned in Section 11, able to encrypt and decrypt for any interpreter are explained and listed.
The first program, KRAKINST.BAS, creates the interpreter-dependent information file KRAKINFO.DAT, which contains the 143 byte Q-key and the number of zeroes at the end of a (plaintext) program. This information file has to be created only once (the program can be considered as an installation program for the second program). The program SAVEs itself into the temporary files TEMPTOK.$$$ (in compressed binary format or tokenized form) and TEMPCODE.$$$ (in encoded binary format). A_i' and B_i are assumed to behave as described in Section 10, so that the Q-key can be computed easily from the bytes (in corresponding positions) in the two temporary files. The number of zeroes at the end of a program is needed to detect the end of a plaintext program. This number is three if the interpreter uses two-byte links and four in the case of three-byte links, and is computed from the position of the low byte of the first line number in the file TEMPTOK.$$$. For this computation the program expects the first line to have line number 10. The second program, KRAK.BAS, uses the information file KRAKINFO.DAT in the encryption or decryption of a (program) file. Whether the program encrypts or decrypts depends on the first byte of the input file (as described in Section 3).

Listing of "KRAKINST.BAS"

```
10 REM  line with line number 10
100 REM ***********************************
110 REM *                                   *
120 REM *  KRAKINST.BAS                     *
130 REM *                                   *
140 REM *                                   *
150 REM *  This program computes the        *
160 REM *  143 bytes long Q-key and         *
170 REM *  the number of zeroes (00-        *
180 REM *  bytes) at the end of a           *
190 REM *  program. These data are          *
200 REM *  stored in the file named         *
210 REM *  "KRAKINFO.DAT".                  *
220 REM *  Note that the first line         *
230 REM *  of this program must have        *
240 REM *  line number 10.                  *
250 REM *                                   *
260 REM *                                   *
270 REM *  Delft, 85.06.13                  *
280 REM *                                   *
290 REM *                                   *
300 REM *  R. van den Assem                 *
310 REM *  W.J. van Elk                     *
320 REM *                                   *
330 REM ***********************************
340 REM ---------------------
350 REM create and open files
360 REM ---------------------
370 SAVE "TEMPTOK.$$$"
380 SAVE "TEMPCODE.$$$", P
390 OPEN "R", #1, "TEMPTOK.$$$", 1
400 FIELD #1, 1 AS PLAIN$
410 OPEN "R", #2, "TEMPCODE.$$$", 1
420 FIELD #2, 1 AS CODE$
430 OPEN "O", #3, "KRAKINFO.DAT"
440 REM ----------
450 REM initialize
460 REM ----------
470 GET #1 : GET #2
480 A%=13 : B%=11
490 REM -------------
500 REM compute Q-key
510 REM -------------
520 FOR I%=1 TO 143
530    GET #1 : GET #2
540    PLAIN% = ASC(PLAIN$)
550    CODE%  = ASC(CODE$)
560    Q1% = (CODE% - B% + 256) MOD 256
570    Q2% = (PLAIN% - A% + 256) MOD 256
580    Q% = Q1% XOR Q2%
590    A% = A%-1 : IF A%=0 THEN A%=13
600    B% = B%-1 : IF B%=0 THEN B%=11
610    PRINT #3, Q%
620 NEXT I%
630 CLOSE #1, #2
640 REM --------------------------
650 REM compute number of zeroes at
660 REM the end of a program
670 REM --------------------------
680 OPEN "R", #1, "TEMPTOK.$$$", 1
690 FIELD #1, 1 AS PLAIN$
700 FOR I%=1 TO 5 : GET #1 : NEXT I%
710 IF ASC(PLAIN$) = 10 THEN NNUL%=4
720 IF ASC(PLAIN$) <> 10 THEN NNUL%=3
730 PRINT #3, NNUL%
740 CLOSE #1
750 REM --------------
760 REM end of program
770 REM --------------
780 CLOSE #3
790 KILL "TEMPTOK.$$$"
800 KILL "TEMPCODE.$$$"
810 END
820 REM ***********************************
```

Listing of "KRAK.BAS"

```
100 REM ***********************************
110 REM *                                   *
120 REM *  KRAK.BAS                         *
130 REM *                                   *
140 REM *                                   *
150 REM *  This program encrypts or         *
160 REM *  decrypts the input file           *
170 REM *  and writes the result into        *
180 REM *  the output file. Whether          *
190 REM *  the program encrypts or           *
200 REM *  decrypts depends on the           *
210 REM *  first byte of the input           *
220 REM *  file. The program needs           *
230 REM *  the data in the file              *
240 REM *  "KRAKINFO.DAT" which can          *
250 REM *  be created by the program         *
260 REM *  "KRAKINST.BAS".                   *
270 REM *                                   *
280 REM *                                   *
```

```
290 REM *  Delft, 85.06.13            *
300 REM *                             *
310 REM *                             *
320 REM *  R. van den Assem           *
330 REM *  W.J. van Elk               *
340 REM *                             *
350 REM *********************************
360 REM -----------------
370 REM read KRAKINFO.DAT
380 REM -----------------
390 DIM Q%(143)
400 OPEN "I",#1,"KRAKINFO.DAT"
410 FOR I%=1 TO 143
420    INPUT #1, Q%(I%)
430 NEXT I%
440 INPUT #1, NNUL%
450 CLOSE #1
460 REM ---------------------
470 REM create and open files
480 REM ---------------------
490 PRINT CHR$(12);
500 INPUT "Name of the input file";NI$
510 OPEN "R",#2,NI$,1
520 FIELD #2, 1 AS I$
530 GET #2
540 IF ASC(I$) > 253 THEN GOTO 580
550 PRINT "File has wrong first byte."
560 CLOSE #2
570 GOTO 490
580 PRINT
590 PRINT "The first byte of the file ";
600 PRINT "indicates that the file "
610 PRINT "has to be ";
620 IF ASC(I$)=254 THEN Z%=0 ELSE Z%=1
630 IF Z%=0 THEN PRINT "decrypted."
640 IF Z%=1 THEN PRINT "encrypted."
650 PRINT
660 INPUT "Name of the output file";NU$
670 OPEN "R",#3,NU$,1
680 FIELD #3, 1 AS U$
690 IF Z%=0 THEN LSET U$ = CHR$(255)
700 IF Z%=1 THEN LSET U$ = CHR$(254)
710 PUT #3
720 REM ----------
730 REM initialize
740 REM ----------
750 A%=13 : B%=11
760 T%=0
770 I%=1
780 REM -------------------
790 REM encrypt or decrypt
800 REM -------------------
810 WHILE T% < NNUL%
820    GET #2
830    X% = ASC(I$)
840    IF Z%=1 THEN 900
850    H% = (X% - B% + 256) MOD 256
860    H% = H% XOR Q%(I%)
870    H% = (H% + A%) MOD 256
880    IF H%=0 THEN T%=T%+1 ELSE T%=0
890    GOTO 940
900    H% = (X% - A% + 256) MOD 256
910    H% = H% XOR Q%(I%)
920    H% = (H% + B%) MOD 256
930    IF X%=0 THEN T%=T%+1 ELSE T%=0
940    LSET U$ = CHR$(H%)
950    PUT #3
960    A%=A%-1 : IF A%=0 THEN A%=13
970    B%=B%-1 : IF B%=0 THEN B%=11
980    I%=I%+1 : IF I%=144 THEN I%=1
990 WEND
1000 REM --------------
1010 REM end of program
1020 REM --------------
1030 CLOSE #2, #3
1040 END
1050 REM *********************************
```

This article by Rene van den Assem and Willem-Jan van Elk is an example of the traditional cryptographic approach used to decrypt enciphered or protected data. The option, **SAVE ⟨filename⟩, P,** in Microsoft's BASIC was never intended as a high level of security, according to a spokesman for the Microsoft Corporation. It was there to prevent very low level snooping. For years, professional programmers, even back in the 8-bit days with a Z80 microprocessor, were aware of methods to overcome this protection scheme. The two short pieces that follow are only two of the methods available to "unprotect" a program. – Harold Joseph Highland, Editor-in-Chief.

A Peek and Poke Approach

In the manual accompanying *Peeks 'n Pokes for the IBM Personal Computer* (a disk to provide users with techniques for getting more out of their PC), Brett Salter included a section for unprotecting BASIC. The version that I have was published in 1982.

The six steps used by Brett Salter to unprotect a BASIC program are:

1. Load BASIC or BASICA using a disk that has at least 128 bytes free.
2. Enter **NEW**.
3. Enter **DEF SEG** with no argument following the statement.
4. Enter **BSAVE "UN.P", 1124, 1** where UN.P is the new name for the program that is to be unprotected.
5. Load the protected program by entering **LOAD "Program.Name"** where the program.name is the protected program.
6. Enter **BLOAD "UN.P", 1124** to unprotect the program.

At this stage the program can be listed or edited under its new name, UN.P.

(*Peeks 'n Pokes* is available from Data Base Decisions, 14 Bonnie Lane, Atlanta GA 30328 USA.)

Harold Joseph Highland

Unprotecting BASIC Using DEBUG.COM

Another method used to unprotect a BASIC program is one that I "borrowed" from a close friend whose computing days go back to the IBM 1620 and who has been involved with microcomputers almost from 1977. Although the method used by Brett Salter is more compact, it does not work with most of the compatibles. This technique works with almost all IBM compatibles but requires the use of BASIC86 which is still available from many user groups.

To unprotect a BASIC program, place DEBUG.COM, BASIC86, and the protected program on a disk. Place that disk in drive B and follow these steps.

> In presenting the following algorithm, all data entered by the user is shown in **bold face type**. The symbol ⟨CR⟩ is used to indicate the pressing of the Return Key.

1. After the system prompt, enter B > **DEBUG BASIC86.COM** ⟨CR⟩
2. The debug prompt, –, will appear and enter:
   – **G9A4** ⟨CR⟩
3. The following copy will now appear on the screen and terminates with the BASIC prompt, *Ok*:
   BASIC-86 Rev. 5.21
   [86-DOS Version]
   Copyright 1977–1981 (C) by Microsoft
   Created: 12-Feb-82
   62367 Bytes free
   OK
4. Load the protected program which for this illustration has been called *Samples*:
   LOAD "SAMPLES" ⟨CR⟩
5. The following register data will appear on the screen and the debug prompt will be at the end of the copy:
   AX = 4600 BX = 0315 CX = 0F0D DX = 0000 SP = FFEC BP = 5692 SI = 0606 DI = 0630 DS = 4448 ES = 4448 SS = 4448 CS = 3D18 IP = 09A4 NV UP DI PL ZR NA PE NC 3D18:09A4 72A4 JB 094A
6. After the debug prompt, enter:
   – **GA72** ⟨CR⟩
7. Again the screen will display register data and the debug prompt:
   AX = 2737 BX = 130A CX = 0C07 DX = 130A SP = FFEA BP = 5692 SI = 130A DI = 130A DS = 4448 ES = 4448 SS = 4448 CS = 3D18 IP = 0A72 NV UP EI PL ZR NA PE NC 3D18:0A72 8B1E3D01 MOV BX,[013D] DS:013D = 0A4D
8. Now change two of the hexadecimal values within the protected program. First enter the first location for the data change:
   – **E630** ⟨CR⟩
9. The following will appear on the screen:
   4448:0630 FE.
10. It is necessary to change the hexadecimal value *FE*; enter **FF**:
    4448:0630 FE.**FF** ⟨CR⟩
11. Now change the second hexadecimal value in the protected program; after the debug system prompt enter:
    – **E584** ⟨CR⟩
12. The following will now appear on the screen:
    4448:0584 FE.
13. Change the *FE* by entering **00** (two zeros):
    4448:0584 FE.**00** ⟨CR⟩
14. Enter the following two commands after the debug prompts when the appear:
    – **MA4D ACD 606** ⟨CR⟩
    – **G** ⟨CR⟩
15. The BASIC *Ok* prompt will appear. Save the unprotected program under a different name, SAMPLESH:
    Ok
    SAVE "SAMPLESH.BAS" ⟨CR⟩
16. Again the BASIC prompt, *Ok*, will appear; leave BASIC by entering **SYSTEM** as would be normal at the conclusion of a BASIC session:
    Ok
    SYSTEM ⟨CR⟩
17. The system now returns to the DEBUG.COM program noting the program has been terminated normally and this is followed by the debug prompt, –. Enter **Q** for quit and this returns us to the system.
    Program terminated normally
    – **Q** ⟨CR⟩

We now have an unprotected program under the name SAMPLESH and it can be edited or listed without protection.

Harold Joseph Highland