Appears in *Proceedings of the 1993 Cambridge Security Workshop,* Springer-Verlag

A Software-Optimized Encryption Algorithm

Phillip Rogaway[1] and Don Coppersmith[2]

[1] IBM Personal Systems, 11400 Burnet Road, Austin TX 78758, USA. e-mail: rogaway@austin.ibm.com.

[2] IBM T.J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, USA. e-mail: copper@watson.ibm.com.

Abstract. We describe a fast, software-oriented, encryption algorithm. Computational cost on a 32-bit processor is about 5 elementary machine instructions per byte of text. The cipher is a pseudorandom function; under control of a key (first pre-processed into an internal table) it stretches a short index into a much longer pseudorandom string. This string can be used as a one-time pad.

1 Introduction

Encrypting Fast In Software. Encryption must often be performed at high data rates, a requirement usually achieved, when at all, with the help of supporting cryptographic hardware. Unfortunately, fast cryptographic hardware is often absent and data confidentiality is sacrificed because the cost of software cryptography is deemed too expensive.

The computational cost of software cryptography is a function of the underlying algorithm and the quality of the implementation of the algorithm. But regardless of implementation, a cryptographic algorithm designed to run well in hardware will not perform in software as well as an algorithm optimized for software execution. The hardware-efficient algorithm of the Data Encryption Standard (DES) is no exception. Since this is the only popular method for symmetric encryption, what is needed is a well-designed, software-optimized encryption method for today's general purpose computers.

To this end, we have designed SEAL (Software Encryption Algorithm; to be known as SEAL 1.0 should other versions arise). It is intended to be used as a stream cipher, providing strong data confidentiality. On a modern 32-bit processor SEAL can encrypt messages at a rate of about 5 instructions per byte. In comparison, the DES algorithm is some 10–30 times as expensive. Even a Cyclic Redundancy Code (CRC) is more costly.

Previous Work. We are not the first to realize the potential value of software-oriented cryptography. In 1991 Merkle described the utility of software-oriented cryptography and he proposed a suite of three software-efficient algorithms [3]. One of them, called "Khufu," uses pre-processing and is similar in spirit to SEAL. In contrast to Khufu, SEAL is not a block cipher—it is instead intended for use as a stream cipher. This is one difference which has helped us to push the speed advantage of SEAL further than Merkle had done in his most aggressive (least number of rounds) instance of Khufu.

An earlier software-oriented block cipher than Khufu is FEAL (Fast Data Encryption Algorithm) [7]. But this algorithm (and variants of it) has not proven to be particularly secure (see [1] for history and attacks). Nor is it all that fast.

RC2 and RC4 are RSA-proprietary algorithms designed by Rivest [6]. The former is a stream cipher which is said to be fast. We have not seen any technical exposition.

2 Characteristics of SEAL

Key characteristics and design choices of SEAL are explained below.

PREPROCESSING THE KEY. In typical applications requiring fast software cryptography, data encryption is required over the course of a communication session to a remote partner, or over the course of a login session to a particular machine. In either case the key a which protects the session is determined at session setup. Typically this session setup takes at least a few milliseconds and is not a time-critical operation. It is therefore acceptable to spend some number of milliseconds at that time to map the (short) key a to a (less concise) representation of the cryptographic transformation specialized to this key. Our cipher has this characteristic.

LENGTH-INCREASING PSEUDORANDOM FUNCTION – VARIABLE OUTPUT AND KEY LENGTHS. The function SEAL is a type of cryptographic object called a "pseudorandom function." Such objects were first defined in [2]. Unlike many pseudorandom functions, SEAL is a length-increasing one: under control of a 160-bit key a, the function maps a 32-bit string n to an L-bit string $\mathsf{SEAL}_a(n)$. The number L can be made as large or as small as is needed for a target application, but output lengths ranging from 512 bytes to 4096 bytes are anticipated. An arbitrary length key a' can index SEAL simply by defining $a = \mathrm{SHA}(a')$.

As a pseudorandom function, $\mathsf{SEAL}_a(\cdot)$ should "look like a random function" if a is random and unknown. The meaning of this is as follows. First a key a is taken at random from $\{0,1\}^{160}$. Next the adversary is given, at random, either a black-box for the function $\mathsf{SEAL}_a(\cdot)$ or else a black-box for a truly random function $\mathcal{R}(\cdot)$. Either maps 32 bits to L bits. The adversary's job is to guess which type of box she has. Say that the adversary *wins* if she correctly guesses "Random" or "Pseudorandom." Our goal is that for any reasonable adversary, she should not win with probability significantly greater than 1/2. Though we will not attempt to define "reasonable" or "significant," we aim to defeat adversaries with substantial computational resources and cleverness.

A pseudorandom function can be used to make a good stream cipher. In a stream cipher the encryption of a message depends not only on the key a and the message x but also on the message's "position" n in the data stream. This position is often a sequence number, present already in the application that would be using the cryptographic method. At other times n may indicate the physical address of a piece of data. In any case the encryption of string x at position n is given by $(n,\ x \oplus \mathsf{SEAL}_a(n))$, where one uses $L = |x|$ bits of output of $\mathsf{SEAL}_a(n)$.

TARGET PLATFORMS. Execution vehicles that should run the algorithm well include the Intel386™, Intel486™, and Pentium™ Processor, and implementations of the IBM Power Architecture™. Because of the particular challenges involved in having a cipher run well on the 386/486, and because of the pervasiveness of this processor family, we optimized with it particularly in mind. By doing well on these difficult-to-optimize-for vehicles we expect to do well on any modern, 32-bit processor.

The relevant limitations of the 386/486/Pentium are: a small register set; a two operand instruction architecture; a small first level cache. In more detail, these chips have the following characteristics. They are 32-bit CISC microprocessors, with current ones running at 25–66 MHz. They have eight general purpose registers, and six segment registers. The instructions generally work on two operands ($A \leftarrow A$ op B) instead of three ($A \leftarrow B$ op C). On the 486, the simplest forms of the instructions `add`, `and`, `mov` and `xor` all take one clock; `ror` (rotate right) takes two. On the 386, each of these numbers is one bigger. Among other assumptions, these instruction counts assume a cache hit, and cache misses can be expensive. The processors use a 4-stage instruction pipeline. As a consequence, if the base register for an address calculation is the destination register of the preceding instruction, an extra cycle will be consumed. Also, use of an index register (as opposed to just a base register) can add an additional cycle. On the 486, use of a non-default segment register may add an additional cycle. The 486 has an 8 Kbyte on-chip cache for data and instructions both; the Pentium Processor has an 8 Kbyte data cache and an 8 Kbyte instruction cache. The Pentium processor has dual pipes, designated U and V, one of which runs a very limited instruction set. It was not a design goal that the cipher should exhibit an instruction dependency structure which would allow us to always fill both pipes.

TABLE-DRIVEN CIPHER. One early decision was whether to make the cipher a straight-line program of logical operations (like MD5 or SHA) or whether to drive it by use of a large table (like Khufu or a software DES), instead. The table-driven approach was selected because we felt that it would lead to a faster and easier-to-design cipher. With the table-driven algorithm we could get very rapid diffusion and there would be little temptation to produce a cipher whose efficient implementation needed self-modifying code.

In view of first-level cache interaction issues and the fact that servers may want to store in second-level cache the representation of the encryption transformation of tens of clients, it was decided that we should not be too generous with the size of the tables that we used. We would settle on a total size for all tables of approximately 3 Kbytes.

procedure $_a(n, \ell,\ \ A, B, C, D,\ n_1, n_2, n_3, n_4)$

$A \leftarrow n \oplus R[4\ell];$
$B \leftarrow (n \ggg 8) \oplus R[4\ell + 1];$
$C \leftarrow (n \ggg 16) \oplus R[4\ell + 2];$
$D \leftarrow (n \ggg 24) \oplus R[4\ell + 3];$

for $j \leftarrow 1$ **to** 2 **do**
  $P \leftarrow A \,\&\, \text{0x7fc};\ B \leftarrow B + T[P/4];\ A \leftarrow A \ggg 9;$
  $P \leftarrow B \,\&\, \text{0x7fc};\ C \leftarrow C + T[P/4];\ B \leftarrow B \ggg 9;$
  $P \leftarrow C \,\&\, \text{0x7fc};\ D \leftarrow D + T[P/4];\ C \leftarrow C \ggg 9;$
  $P \leftarrow D \,\&\, \text{0x7fc};\ A \leftarrow A + T[P/4];\ D \leftarrow D \ggg 9;$

$(n_1,\ n_2,\ n_3,\ n_4) \leftarrow (D,\ B,\ A,\ C);$

$P \leftarrow A \,\&\, \text{0x7fc};\ B \leftarrow B + T[P/4];\ A \leftarrow A \ggg 9;$
$P \leftarrow B \,\&\, \text{0x7fc};\ C \leftarrow C + T[P/4];\ B \leftarrow B \ggg 9;$
$P \leftarrow C \,\&\, \text{0x7fc};\ D \leftarrow D + T[P/4];\ C \leftarrow C \ggg 9;$
$P \leftarrow D \,\&\, \text{0x7fc};\ A \leftarrow A + T[P/4];\ D \leftarrow D \ggg 9;$

Fig. 1. *Initialization of $(A, B, C, D,\ n_1, n_2, n_3, n_4)$ from (n, ℓ). This initialization depends on a-derived tables T and R.*

3 Definition of SEAL

Notation. We call a 32-bit string a "word" and an 8-bit string a "byte." The empty string is denoted λ. The bits of a length-t string x will be numbered $x_0x_1 \ldots x_{t-1}$. We write numbers in hexadecimal by preceding them with "0x" and then using the symbols "a"–"f" to represent decimal numbers 10–15, respectively. By $y \ggg t$ we denote a right circular shift of the word y by t bits; in other words, the i-th bit of $y \ggg t$ is $y_{(i-t) \bmod 32}$. By "$\wedge$" "$\vee$" and "$\oplus$" we denote bitwise AND, OR, and XOR; by $\overline{A}$ we denote the complement of A. By $A + B$ we denote the sum, ignoring the carry, of the unsigned integers A and B; this is of course the sum mod 2^{32} of numbers A and B. By "$\|$" we denote the concatenation operator. By $odd(\cdot)$ we mean the predicate which is true if and only if its argument is an odd number.

Output Length. Recall that we think of SEAL as producing variable length output. Let L be the number of output bits desired. We assume a large bound on L, say $L \leq 64 \cdot 1024 \cdot 8$. Our algorithm stops generating bits as soon as it produces L' of them, where L' is the least multiple of 128 which is at least L.

Mapping The Key To The Tables. Our first task is to specify the tables T, R, and S, all of which depend only on a. The key a is used only to define these three tables.

function $_a(n)$

$y = \lambda;$

for $\ell \leftarrow 0$ **to** ∞ **do**

$\quad {}_a(n, \ell,\ A, B, C, D,\ n_1, n_2, n_3, n_4);$

$\quad$ **for** $i \leftarrow 1$ **to** 64 **do**

$\quad\quad P \leftarrow A \,\&\, \text{0x7fc};\quad B \leftarrow B + T[P/4];\ A \leftarrow A \rangle\rangle\rangle\, 9;\ B \leftarrow B \oplus A;$
$\quad\quad Q \leftarrow B \,\&\, \text{0x7fc};\quad C \leftarrow C \oplus T[Q/4];\ B \leftarrow B \rangle\rangle\rangle\, 9;\ C \leftarrow C + B;$
$\quad\quad P \leftarrow (P + C) \,\&\, \text{0x7fc};\ D \leftarrow D + T[P/4];\ C \leftarrow C \rangle\rangle\rangle\, 9;\ D \leftarrow D \oplus C;$
$\quad\quad Q \leftarrow (Q + D) \,\&\, \text{0x7fc};\ A \leftarrow A \oplus T[Q/4];\ D \leftarrow D \rangle\rangle\rangle\, 9;\ A \leftarrow A + D;$

$\quad\quad P \leftarrow (P + A) \,\&\, \text{0x7fc};\ B \leftarrow B \oplus T[P/4];\ A \leftarrow A \rangle\rangle\rangle\, 9;$
$\quad\quad Q \leftarrow (Q + B) \,\&\, \text{0x7fc};\ C \leftarrow C + T[Q/4];\ B \leftarrow B \rangle\rangle\rangle\, 9;$
$\quad\quad P \leftarrow (P + C) \,\&\, \text{0x7fc};\ D \leftarrow D \oplus T[P/4];\ C \leftarrow C \rangle\rangle\rangle\, 9;$
$\quad\quad Q \leftarrow (Q + D) \,\&\, \text{0x7fc};\ A \leftarrow A + T[Q/4];\ D \leftarrow D \rangle\rangle\rangle\, 9;$

$\quad\quad y \leftarrow y \ \|\ B + S[4i{-}4] \ \|\ C \oplus S[4i{-}3] \ \|\ D + S[4i{-}2] \ \|\ A \oplus S[4i{-}1];$

$\quad\quad$ **if** $|y| \geq L$ **then return** $(y_0 y_1 \ldots y_{L-1});$

$\quad\quad$ **if** $odd(i)$ **then** $(A,\ C) \leftarrow (A + n_1,\ C + n_2)$
$\quad\quad\quad\quad$ **else** $(A,\ C) \leftarrow (A + n_3,\ C + n_4);$

Fig. 2. *Cipher mapping 32-bit index n to L-bit string* $_a(n)$ *under the control of a-derived tables T, R, and S.*

We specify the tables using a function G. For a a 160-bit string and i an integer, $0 \leq i < 2^{32}$, $G_a(i)$ is a 160-bit value. The function G is based directly on the Secure Hash Algorithm (SHA) [5]. Its definition is given in Appendix A.

Let us re-index G to construct a function Γ whose images are 32-bit words instead of 160-bit ones. The function Γ is defined by $\Gamma_a(i) = H^i_{i \bmod 5}$ where $H^i_0 H^i_1 H^i_2 H^i_3 H^i_4 = G_a(\lfloor i/5 \rfloor)$. Thus a table of Γ-values is exactly a table for G-values read left-to-right, top-to-bottom.

Now define

$$\begin{array}{ll} T[i] = \Gamma_a(i) & \text{for all } 0 \leq i < 512, \\ S[j] = \Gamma_a(\text{0x1000} + j) & \text{for all } 0 \leq j < 256, \text{ and} \\ R[k] = \Gamma_a(\text{0x2000} + k) & \text{for all } 0 \leq k < 4\lceil (L-1)/8192 \rceil. \end{array}$$

We note that the computation of these tables requires calculating the compression function of the SHA on a variable number of blocks—131 times to get an output length of 512 bytes, 207 times for the maximal output length of 64 Kbytes.

THE PSEUDORANDOM FUNCTION. Given the number L, the tables T, R, and S determined by a, and a 32-bit index n, the algorithm of Figure 2 stretches n to an L-bit pseudorandom string y.

The algorithm uses a routine Initialize to map n and ℓ to the words A, B, C, D, n_1, n_2, n_3, n_4. That procedure is given in Figure 1.

The outer loop of Figure 2 is to be broken by line 10 when enough output bits have been collected.

We comment that all those divisions by 4 in the algorithm description aren't really to be thought of as divisions or even as shifts; we are just indexing into the table in units of bytes, instead of units of words. This is more efficient on some platforms and no less efficient on any byte-addressable execution vehicles.

4 Performance

An assembly implementation with $L = 512 \times 8$ bits and the key already mapped into tables encrypts at a rate of 3.6 Mbytes/second on an IBM PS/2™ with a 25 MHz 486. A PC with a 50 MHz 486 ran the same code at 7.2 Mbytes/second. A C-language implementation performing the same computation encrypts at 4.5 Mbytes/second on a RISC System/6000™ Model 530. (This is a 40 MHz machine.) These experiments compute $\oplus_{i=0}^{M}\mathsf{SEAL}_a(i)$ for a large M.

5 Concluding Remarks

Many applications which require fast software encryption also require fast message authentication. For such applications SEAL can be combined with a software optimized message authentication code (MAC). An efficient software MAC is the subject of separate research.

A cipher very similar to SEAL can be given for 64-bit architectures. The tables would be twice as wide and Initialize would be slightly changed.

As is customary in introducing a new primitive, we call for and encourage attacks.

References

1. E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, 1993.
2. O. Goldreich, S. Goldwasser and S. Micali, "How to construct random functions." *Journal of the ACM,* Vol. 33, No. 4, 210–217. 1986.
3. R. Merkle, "Fast software encryption functions." Proceedings of CRYPTO '91, Springer-Verlag. Also U.S. patent 5,003,597. March 1991.
4. National Bureau of Standards, "Data Encryption Standard." Federal Information Processing Standards Publication 46. January 1977.
5. National Institute of Standards, FIPS Publication 180, "Secure Hash Standard." May 1993.
6. RSA Data Security, Inc., promotional materials and personal communications. Redwood City, California.
7. A. Shimizu and S. Miyaguchi, "Fast Data Encryption Algorithm FEAL." Eurocrypt '87.

Appendix A. The Table-Generation Function

We specify $G_a(i)$ for 160-bit string a and integer $0 \le i < 2^{32}$. The latter is treated as a 32-bit string whose value as an unsigned binary number is i. This function is defined directly from Sections 5–7 of [5]; the definition is repeated here only for ease of reference.

First we make the following definitions. For $0 \le t \le 19$, set $K_t = \text{0x5a827999}$ and $f_t(B,C,D) = (B \wedge C) \vee (\overline{B} \wedge D)$. For $20 \le t \le 39$ set $K_t = \text{0x6ed9eba1}$ and $f_t(B,C,D) = B \oplus C \oplus D$. For $40 \le t \le 59$ set $K_t = \text{0x8f1bbcdc}$ and $f_t(B,C,D) = (B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$. For $60 \le t \le 79$ set $K_t = \text{0xca62c1d6}$ and $f_t(B,C,D) = B \oplus C \oplus D$.

The 160-bit string a is broken up into five 32-bit words, $a = H_0H_1H_2H_3H_4$, and the 512-bit M_1 is set to $i \parallel 0^{480}$ and then processed by:

a. Divide M_1 into 16 words $W_0, W_1, \ldots, W_{15}$ where W_0 is the left-most word, so that $W_0 = i$, $W_1 = W_2 = \ldots = W_{15}$.
b. For $t = 16$ to 79 let $W_t = W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}$.
c. Let $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$.
d. For $t = 0$ to 79 do
$$\text{TEMP} = A \rangle\rangle\rangle\, 27 + f_t(B,C,D) + E + W_t + K_t$$
$$E = D;\ D = C;\ C = B \rangle\rangle\rangle\, 2;\ B = A;\ A = \text{TEMP};$$
e. $H_0 = H_0 + A$; $H_1 = H_1 + B$; $H_2 = H_2 + C$; $H_3 = H_3 + D$; $H_4 = H_4 + E$;

After processing M_1 the value of $G_a(i)$ is the 160-bit string $H_0H_1H_2H_3H_4$.

Appendix B. Test Case

The purpose of this appendix is to provide adequate data to verify a correct implementation of SEAL. So suppose the key is the 160-bit string

$a =$ `67452301 efcdab89 98badcfe 10325476 c3d2e1f0`

and assume we want SEAL to produce 4 Kbyte outputs (i.e., $L = 32768$ bits). Then the table R consists of words $R[0]$, $R[1], \ldots$, $R[15]$:

```
e55c4955 59a40fb1 495a4508 850454f4 5b0200e9 8594d576 dc10d31e 1ecdb16b
e699bcf5 e2d35068 300f8bdc bd39f67c c06491dc 119fd703 2db7fdf7 2fee5c42
```

The table T consists of words $T[0]$, $T[1], \ldots$, $T[511]$:

```
92b404e5 56588ced 6c1acd4e bf053f68 09f73a93 e786e543 b184176b 081df5cc
da3a23e0 a2a758ea e85e0feb 078b776d a13956cb 9e7c092c 6970262b d1472567
........ ........ ........ ........ ........ ........ ........ ........
ece03c22 82031f4d c09f5ed8 88e489f0 61c376e7 2056695a 814a313d e48f7e71
```

The table S consists of words $S[0]$, $S[1], \ldots$, $S[255]$:

```
68318e58 0781c371 8bf691be 8afd6893 1a421779 bdfec86e 8382b844 da3ecc73
bc20de5c 5755b8fd a82fe7da e2c59ebc 3c43fbd0 db005522 64ee8a5e 97553175
........ ........ ........ ........ ........ ........ ........ ........
f1ab9dab 72c8d7c9 57222650 e41fc10a 473af4e3 6364b447 c173fe02 8cbe7fbf
```

Letting n = `013577af`, the string $y = \mathsf{SEAL}_a(n)$ consists of $y[0]y[1]\cdots y[1023]$ (the XOR of these 1024 words being `e421eec0`):

```
9c3cf4eb 56ec2dd2 cd663283 eaf8b9f7 2eb4fd0f f5bc9ebb 11429760 6555325a
e3adc16f caf41b01 972186cf 07e119c3 35050126 c9c6a240 15f90e07 16611570
0bd13fc5 b0bbb36d 7f66916f c92ed8c2 a75d7b65 2c2e92f8 25c55f8d c8260443
50eaee36 f57610ee 92602e91 4fdf9668 eb540c64 9160c136 47f77586 94e58747
b3a4eaaa 6a454573 70fba3a9 a390caba 6789c508 726621ed e4133cca 3c39a962
0f93f38f 1aa8e9f6 c99b9d09 c67f0822 20c996d2 43ad4c2d c1a7d3b2 0c8686ef
8cdbb45b 5b2b59ae d634d2de 925368b9 ce376564 af0b0f2b c70507e0 373dfbc0
64a4270a 03ec3e43 555bba96 53f3b90d 18ca353d dba6bb48 4b5a20ce d003c675
d943162e 4093e3d8 7e7cf817 c3f76a81 8128e2a4 cccc4f13 fc241ebb 87ba5da0
969b22a2 8dfaf964 bec9573f 1143255c 3c75f42e 19eefbc7 1f3b3aad 66400bfc
3ca899b3 917828a8 d8b1aa3e 8a8b049f b2bf6441 7be6c1e2 f7415160 24264ad6
c7af4fa2 231f8764 f8bdd20e 760b0cc8 e8fa91e1 51fbf403 fb4edcc3 c6b727bb
feb29efd a7c847d8 5f122847 f4975e56 8d8722a9 0a67702a f6cdebd9 e1ad6bdd
6b2fe6f9 80f2a776 fec2550b c49d9638 edb7e43e 1ea22586 c1c184f5 0349c93e
f7cc4e73 cbc5d581 b838425a a8d5e544 22dd84b2 691b4672 d4dc9852 88ba4e0c
b472e60a 8ed0ee05 38695e83 ae68fe11 63c8c25e 92fd803a a477d7ec 65eea731
f81cdbb6 0cdbade3 cacd8914 f63a02b7 8083234b 96f3b7e5 6ebc8e9b 3571f02d
ca2f5f3a 8a85d550 4b7705ea 43e503af 6d6db5e3 127fd171 feb7d0d0 b7266434
61b316f6 a2d82105 ab2e2b5e 2e40c41a a1b059b3 01c2f977 3b526317 550fa3be
75d3a064 4ad93f0c 8d586701 659cc5ff 74c63d51 ff9f54b8 131185f2 2afd8814
84a6e34a fa7af37c c0319633 dec46921 0724320f afac79d8 6373edc9 264421fa
36555173 e4e9758c ccce0e6c d1c9be99 d7dcee9f e1ce2ea1 c0b1582e 336ccc6b
a8cafbc2 be54edae 6ffff7cc 01e3f27f 279d43f0 b223411a c764ed5f e33af06d
3cc9cf21 329ca864 0cb34429 fbaba601 4a08b27a 2198fbda 4336d3e5 6cf34c6b
a3c1cf0a b39ea8f7 dc867345 3b2271ba 9ec8b148 f59a1a9b 51dc5f27 0f9b337e
b4a544d4 59749976 712ef2f2 6fcbed5d 18df76f2 ee20e250 d92f376f d97efaf1
7a14e5e4 9e2ac9bd 9f5b764c 70f2ef3c 282fdcf7 ff572655 f28e093b c080b736
5ce89c3e 459d37af a189c13d 8d1b55b8 2e7922c8 840e4072 70e3e497 bbdab75f
fa47fdca e7760d48 bb243dec bddcd8e0 83d04879 eb8cc529 cb66b4b5 92103a7b
9a05ebf0 638a6eb3 c6fffaf3 9c817b53 286e8c06 e048e226 634ee8cd b39b9507
cf418ebf 560a13dd eab6b5d7 d368f422 5d87f4f8 70a87bc9 a22688bc 08b49f7c
66f5822c 1a7d26b2 32227ef7 2b5cba59 e8d8e3b8 ce9a0bab 075aea70 9af456ed

d2f91c98 f7573abd c90e510f 8bbede45 4b3db413 eb90f47b d6c05db1 8c437e10
ac58339a 69c8a201 089c412e 9c5ea298 c1a4bdf4 3a4088a0 cb1d7f46 93c3df3d
........ ........ ........ ........ ........ ........ ........ ........
da9f4c73 c0b7e6f5 aeb46849 5dd72e83 b5770d75 e18c7336 f620ef9c b2d84735
```

Intel, Intel386, Intel486 and Pentium are trademarks of Intel Corporation. IBM, PS/2, and RISC System/6000 are trademarks of International Business Machines Corporation.