Full version of [16]. Last revised September 18, 1997.

A Software-Optimized Encryption Algorithm

Phillip Rogaway[1] and Don Coppersmith[2]

[1] Department of Computer Science, Engineering II Building, University of California, Davis, CA 95616, U.S.A. rogaway@cs.ucdavis.edu

[2] IBM T.J. Watson Research Center, PO Box 218, Yorktown Heights, NY 10598, U.S.A. copper@watson.ibm.com

Abstract. We describe a software-efficient encryption algorithm named SEAL 3.0. Computational cost on a modern 32-bit processor is about 4 clock cycles per byte of text. The cipher is a pseudorandom function family: under control of a key (first pre-processed into an internal table) it stretches a 32-bit position index into a long, pseudorandom string. This string can be used as the keystream of a Vernam cipher.

Key words. Cryptography, Encryption, Fast encryption, Pseudorandom function family, Software encryption, Stream cipher.

1 Introduction

ENCRYPTING FAST IN SOFTWARE. Encryption must often be performed at high data rates, a requirement sometimes met with the help of supporting cryptographic hardware. Unfortunately, cryptographic hardware is often absent and data confidentiality is sacrificed because the cost of software cryptography is deemed to be excessive.

The computational cost of software cryptography is a function of the underlying algorithm and the quality of its implementation. 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-oriented Data Encryption Algorithm (DES) is no exception. Often 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). SEAL is a pseudorandom function family: under control of a key, first preprocessed into a set of tables, SEAL stretches a 32-bit "position index" into a keystream of essentially arbitrary length. One then encrypts by XORing this keystream with the plaintext, in the manner of a Vernam cipher. As with any Vernam cipher it is imperative that the keystream only be used once.

On a modern 32-bit processor SEAL can encrypt messages at a rate of about 4 clock cycles per byte of text. In comparison, the DES algorithm is more than 10 times as expensive. Even a Cyclic Redundancy Code (CRC) is more costly.

Related Work. We are not the first to realize the value of software-optimized cryptography. In 1991 Merkle described the utility of software-oriented cryptography and he proposed a suite of three software-efficient algorithms [8]. One of them, called "Khufu," is a block cipher which is similar in spirit to SEAL.

An earlier software-oriented block cipher than Khufu is FEAL [18]. But this algorithm and its variants have not proven to be particularly secure (see [1] for history and attacks). Nor is it all that fast.

RC4 is a popular, software-efficient stream cipher designed by Rivest [13]. It is fast, though less fast than SEAL. RC5 is a software-efficient block cipher. It too was designed by Rivest [14]. Some other software-efficient ciphers include Blowfish[17] and WAKE [19].

History And Naming. The full name of the cipher described in this paper is SEAL 3.0. An earlier version of this cipher was described in 1993 [16] and denoted SEAL 1.0. Though SEAL 3.0 is the first modification to SEAL 1.0 which the authors have described, a variant known as SEAL 2.0 had already appeared in the literature [7]: it was identical to SEAL 1.0 apart from using NIST's revised Secure Hash Algorithm (SHA-1) instead of the original one (SHA) [10]. While SEAL 3.0 retains that change, the more significant adjustment is responsive to an attack by Handschuh and Gilbert [5]. See Section 5 for further information on their attack and the differences between SEAL 3.0 and SEAL 1.0.

In this paper the name SEAL, by itself, always refers to SEAL 3.0.

2 Characteristics of the Cipher

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, in most applications, to spend some number of milliseconds to map the (short) key a to a (less concise) representation of the cryptographic transformation specialized to this key. Our cipher has this characteristic. As such, SEAL is an inappropriate choice for applications which require rapid key-setup.

Length-Increasing Pseudorandom Function – Variable Output And Key Lengths. The function SEAL is a type of cryptographic object called a *pseudorandom function family* (PRF). Such objects were first defined in [4]. SEAL is a *length-increasing* PRF: under control of a 160-bit key a, SEAL maps a 32-bit string n to an L-bit string $\mathsf{SEAL}(a, n, L)$. The number L can be made as large or as small as is needed for a target application, but output lengths ranging from a few bytes to a few thousand bytes are anticipated. An arbitrary length key a' can be used as the key for SEAL simply by selecting $a = \text{SHA-1}(a')$.

As a pseudorandom function family, $\mathsf{SEAL}(a, \cdot, L)$ 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 $\mathsf{SEAL}(a, \cdot, L)$ 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 PRF 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 counter (sequence number) which indicates which message is being enciphered. The encryption of string x at position n is given by $\langle n,\ x \oplus \mathsf{SEAL}(a, n, L)\rangle$, where $L = |x|$. In other applications n might indicate the address of a piece of data on disk.

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

Some of the relevant limitations of the 386/486/Pentium are a small register set, a two-operand instruction architecture, and a small first level cache. Here is some further detail which was important in design choices. These processors have 8 general registers (including the register normally used as a stack pointer). Most instructions work on two operands ($A \leftarrow A$ op B) instead of three ($A \leftarrow B$ op C). The 486 has an 8 KByte on-chip cache for data and instructions, while the Pentium has an 8 KByte data cache and an 8 KByte instruction cache. Cache misses can be expensive. The 486 and Pentium processors use a 5-stage instruction pipeline, and if the base register for an address calculation is the destination register of the preceding instruction, an extra cycle will be consumed. The Pentium processor has dual instruction pipes, one of which runs a very limited instruction set. It was not a design goal for the cipher to 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 [12] or SHA-1 [10]) or to drive it instead by the 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 less temptation to produce a cipher whose most efficient implementation used self-modifying code.

In view of the size of the first-level cache, 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 3–4 KBytes.

procedure $\mathsf{Initialize}(n, \ell,\ \ A, B, C, D,\ \ n_1, n_2, n_3, n_4)$

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

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

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

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

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 the Cipher

NOTATION. We call a 32-bit string a "word" and an 8-bit string a "byte." The empty string is denoted λ. 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 $y \rangle\rangle\rangle\, t$ we denote a right circular shift of the word y by t bits; in other words, the i-th bit of $y \rangle\rangle\rangle\, t$ is $y_{(i-t) \bmod 32}$. Similarly, $y \langle\langle\langle\, t$ denotes a left circular shift of y by t bits. By "&" "$\vee$" 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 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.

```
function SEAL(a, n, L)
y = λ;
for ℓ ← 0 to ∞ do
  Initialize(n, ℓ,  A, B, C, D,  n1, n2, n3, n4);
  for i ← 1 to 64 do
    P ← A & 0x7fc;        B ← B + T[P/4]; A ← A ⋙ 9; B ← B ⊕ A;
    Q ← B & 0x7fc;        C ← C ⊕ T[Q/4]; B ← B ⋙ 9; C ← C + B;
    P ← (P + C) & 0x7fc;  D ← D + T[P/4]; C ← C ⋙ 9; D ← D ⊕ C;
    Q ← (Q + D) & 0x7fc;  A ← A ⊕ T[Q/4]; D ← D ⋙ 9; A ← A + D;

    P ← (P + A) & 0x7fc;  B ← B ⊕ T[P/4]; A ← A ⋙ 9;
    Q ← (Q + B) & 0x7fc;  C ← C + T[Q/4]; B ← B ⋙ 9;
    P ← (P + C) & 0x7fc;  D ← D ⊕ T[P/4]; C ← C ⋙ 9;
    Q ← (Q + D) & 0x7fc;  A ← A + T[Q/4]; D ← D ⋙ 9;

    y ← y || B+S[4i−4] || C⊕S[4i−3] || D+S[4i−2] || A⊕S[4i−1];

    if |y| ≥ L then return (y0 y1 ... y_{L−1});

    if odd(i) then (A, B, C, D) ← (A + n1, B + n2, C ⊕ n1, D ⊕ n2)
              else (A, B, C, D) ← (A + n3, B + n4, C ⊕ n3, D ⊕ n4);
```

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

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 $L \le 64 \cdot 1024 \cdot 8$. So at most 64 KBytes may be produced per index.

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

We specify the tables using a function G. For a a 160-bit string and i an integer, $0 \le i < 2^{32}$, $G_a(i)$ is a 160-bit value. The function G is just the compression function of the Secure Hash Algorithm SHA-1 [10]. For completeness, 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 $H_0^{5j} \| H_1^{5j+1} \| H_2^{5j+2} \| H_3^{5j+3} \| H_4^{5j+4} = G_a(j)$, for $j = \lfloor i/5 \rfloor$.

Thus a table of Γ-values is exactly a table for G-values read left-to-right, top-to-bottom.

Now define

$$
\begin{array}{ll}
T[i] = \Gamma_a(i) & \text{for all } 0 \le i < 512, \\
S[j] = \Gamma_a(\mathrm{0x1000} + j) & \text{for all } 0 \le j < 256, \text{ and} \\
R[k] = \Gamma_a(\mathrm{0x2000} + k) & \text{for all } 0 \le k < 256.
\end{array}
$$

Four words of the array R are required for each kilobyte (or fraction of a kilobyte) of $\mathsf{SEAL}(a, n, L)$. Thus if one has a bound $L_{\max}$ on the maximal possible value of L then it is adequate to compute $R[k]$ for $0 \le k < 4\lceil L_{\max}/8192\rceil$. For the maximal permitted output length of 64 KBytes one needs to calculate the SHA-1 compression function 207 times.

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

The algorithm uses a routine Initialize which, using tables T and R, maps 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.

TERMINOLOGY. For purposes of subsequent discourse, a *round* refers to the execution of any one of lines 1–8 in Figure 2, while an *iteration* is the execution of all of the lines (1–11) associated to a given value of i. Thus there are eight rounds in each iteration.

4 Explanations and Design Heuristics

Some of the structure of SEAL may be made less mysterious by the general explanations of this section and the specific attacks of Section 6. The following general heuristics were employed:

1 Using a large, secret, key-derived "S-box" (the 2 KByte table T).

2 Alternating arithmetic operations which don't commute (addition mod 2^{32} and bitwise XOR).

3 Using internal state maintained by the cipher and not directly manifest in the output data stream (the registers n_1, n_2, n_3, n_4).

4 Using simple, well-known methods where adequate (using SHA-1 to generate the tables).

Somewhat more specific heuristics:

5 Varying the round function according to the round number (e.g., alternating use of P and Q.)

6 Varying the iteration function according to the iteration number (e.g., (n_1, n_2) or (n_3, n_4) in line 11; and $S[\cdot]$-values associated to the iteration).

The attention to the parity of the round and iteration number may help against attacks which play off successive rounds or successive iterations.

Details of the method used to produce the tables T, R and S (the use of SHA-1, the indexing method, etc.) are not believed to be particularly important; we think of these tables as "random" (no design rules are built into their construction) and we expect that any good pseudorandom generator applied to the key should work fine.

Details of the function Initialize are believed to be of secondary importance. We want A, B, C, D, n_1, n_2, n_3 and n_4 to be unpredictable functions (n, ℓ).

Each of the final instruction on lines 1–4 helps to diffuse information in A, B, C and D. An earlier version of the cipher made analogous register modifications in lines 5–8 but the statements would seem to have less value there and so they were removed to save cost.

Some performance-related explanations are given below:

1 The divisions by 4 are not to be implemented by divisions or shifts; we are simply indexing into T in units of bytes instead of words. This is more efficient on some platforms (which may penalize for "scaling" word offsets) and no less efficient on any platform we considered. In a high-level language these divisions might be implemented as a cast.

2 On all processors we know of there is no performance difference between using addition and XOR, and so there is no performance reason to favor the latter.

3 On our target two-operand machine architectures it is the same cost to compute $P \leftarrow (P+A)$ & 0x7fc and then fetch $T[P/4]$ as it would be to fetch $T[(A$ & 0x7fc$)/4]$. This is because the computation of $T[(A$ & 0x7fc$)/4]$, to preserve A, must begin by moving A into a temporary register. That move is the same cost as adding A to register P.

4 The state of P and Q is not maintained across iterations simply because machines with only 8 registers will need to use the registers holding P and Q at the end of the iteration. We did not want to spend the extra cycles to write P and Q to memory and then read them back.

5 Operations are arranged so that in the clock cycle immediately following a table lookup there is always something worthwhile to do which does not depend on the value which is retrieved.

5 Design Process

A brief description of the design process which has led to SEAL may be considered relevant or interesting to some.

SEAL 1.0. The project began in the summer of 1992 in response to the perception of increasing customer needs for software-efficient cryptography. Goals of the design were first enumerated in a presentation of October 1992. Goals evolved as we learned more; there was never any fixed or formal statement of requirements.

Merkle's cipher Khufu was identified as the most relevant prior art. We chose it as our starting point and searched for ways that would lead to something even faster.

A design "philosophy" emerged. We thought it better to do exceptionally well in environments having a particular set of minimal environmental characteristics than to do reasonably well across a wider range of environments. Our chosen set of operating characteristics became: a 32-bit machine with at least eight general purpose registers, a cache of at least 8 KBytes, and a usage scenario which partitions encryption into a performance non-critical key setup followed by repeated and performance-critical encipherment of a reasonably large number of bytes.

We didn't care about the syntactic flavor of the cipher we would produce—even whether it was a block cipher or something else seemed irrelevant, except insofar as this might influence the cipher's speed.

The first suggestion (March 1993) was for a block cipher, but soon we developed a basic "structure" for a pseudorandom function family which was going to be faster. This structure consisted of having four registers (A, B, C, D), each of which would modify a "neighboring" register as a result of a single lookup in a key-derived table. After some number of such register modifications we would "peel off" the current value of the four registers and append them to the growing keystream. This process would then be repeated.

A total of nine designs were considered between March 1993 and October 1993. Each revision was aimed to improve speed or perceived strength. Rogaway would prepare a specification and Coppersmith would attack it. Attacks were considered far enough to make clear what was their main idea, not to assess their exact efficacy. Rogaway would then study the attack, try to identify some essential weakness it exploited, and then modify the cipher (without decreasing its speed) to try to foil any similar cleverness.

The inner loop (Figure 2) was the subject of almost all of our effort. Very little attention was paid to Initialize (Figure 1) or to the table generation method.

The design progressed entirely on paper. No statistical tests or other experiments were performed during the design of the cipher. Our proposal, SEAL 1.0, was first described in December 1993 [16].

SEAL 3.0. In 1996 Handschuh and Gilbert [5] described an attack on a simplified version of SEAL 1.0, and an attack on SEAL 1.0 itself. They require about 2^{30} "samples," each 4-words long, to distinguish SEAL 1.0 from a random function. Their attack is responsible for the main change between SEAL 1.0 and SEAL 3.0. That change requires the use of two new XORs for each 4 words of output, as we now explain.

Refer to Line 11 of Figure 2. The corresponding line in SEAL 1.0 had been: **if** $odd(i)$ **then** $(A,\ C) \leftarrow (A+n_1,\ C+n_2)$ **else** $(A,\ C) \leftarrow (A+n_3,\ C+n_4)$. Now we modify all four registers, A, B, C, D, instead of just the two registers A, C. This better obscures relationships between the (A, B, C, D) and (A', B', C', D') values of successive iterations. Without the change there is a useful property on (D, C', D'), say, which does not depend on any of n_1, n_2, n_3, n_4; see [5]. Unpublished predecessors of SEAL 1.0 resembled SEAL 3.0 in modifying each of (A, B, C, D) at the end of an iteration; removing the modifications to B and D was a poorly-chosen optimization.

The other difference between SEAL 3.0 and SEAL 1.0 is that in SEAL 3.0 (and SEAL 2.0) table generation uses SHA-1 in lieu of the older SHA.

STATISTICAL TESTS. In response to a referee's request we subjected SEAL to a battery of statistical tests developed by Marsaglia [6]. We computed the 10 MByte string $y = \mathsf{SEAL}(a, 0, L) \| \mathsf{SEAL}(a, 1, L) \| \cdots \| \mathsf{SEAL}(a, 156249, L)$ for a fixed key a and $L = 64 \cdot 8$ (i.e., 64 bytes). None of the 15 tests in [8] revealed statistical anomalies in y. In a second experiment we computed the 10.03 MByte string $z = \mathsf{SEAL}(a, 0, L) \| \mathsf{SEAL}(a, 1, L) \| \cdots \| \mathsf{SEAL}(a, 152, L)$, where $L = 64 \cdot 1024 \cdot 8$ (i.e., 64 KBytes). Again, none of the 15 tests revealed statistical anomalies in z.

6 Illustrative Attacks

This section illustrates some attack ideas which were important to SEAL's evolution. We describe three attacks on a simplified version of our cipher. This simplified cipher, WEAK, is show in Figure 3.

function $\mathsf{WEAK}(a, n)$

$y \leftarrow \lambda;$

$\mathsf{Initialize}_a(n, 0, \;\; A, B, C, D, \;\cdots);$

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

$\quad P \leftarrow A \;\&\; \mathrm{0x1ff};\; B \leftarrow B \oplus T[P];\; A \leftarrow A \rangle\rangle\rangle 9;\; B \leftarrow B \oplus A;$
$\quad P \leftarrow B \;\&\; \mathrm{0x1ff};\; C \leftarrow C \oplus T[P];\; B \leftarrow B \rangle\rangle\rangle 9;\; C \leftarrow C \oplus B;$
$\quad P \leftarrow C \;\&\; \mathrm{0x1ff};\; D \leftarrow D \oplus T[P];\; C \leftarrow C \rangle\rangle\rangle 9;\; D \leftarrow D \oplus C;$
$\quad P \leftarrow D \;\&\; \mathrm{0x1ff};\; A \leftarrow A \oplus T[P];\; D \leftarrow D \rangle\rangle\rangle 9;\; A \leftarrow A \oplus D;$
$\quad P \leftarrow A \;\&\; \mathrm{0x1ff};\; B \leftarrow B \oplus T[P];\; A \leftarrow A \rangle\rangle\rangle 9;$
$\quad P \leftarrow B \;\&\; \mathrm{0x1ff};\; C \leftarrow C \oplus T[P];\; B \leftarrow B \rangle\rangle\rangle 9;$
$\quad P \leftarrow C \;\&\; \mathrm{0x1ff};\; D \leftarrow D \oplus T[P];\; C \leftarrow C \rangle\rangle\rangle 9;$
$\quad P \leftarrow D \;\&\; \mathrm{0x1ff};\; A \leftarrow A \oplus T[P];\; D \leftarrow D \rangle\rangle\rangle 9;$
$\quad y \leftarrow y \;\|\; B \oplus S[4i-4] \;\|\; C \oplus S[4i-3] \;\|\; D \oplus S[4i-2] \;\|\; A \oplus S[4i-1];$

return y;

Fig. 3. *The cipher* WEAK, *attacks on which are given in the text. Under the control of a-derived tables T, R and S (computed exactly as with* SEAL*) this cipher maps 32-bit position index n to 256-word string* $\mathsf{WEAK}(a, n)$.

ASSEMBLE A LIST OF $T[\alpha]$ op $T[\beta]$ VALUES. A simple attack on WEAK is based on the observation that each of A, B, C and D is modified only two times using T, and the net-change due to this pair of T-dependent modifications is almost directly visible to the adversary.

In this and all subsequent attacks we fix an (unknown) key a and provide the adversary sample output strings, each of the form $y = \mathsf{WEAK}(a, n)$. The adversary will not need to know the n which produced each string y.

Fix one of the strings y the adversary collects and let us write $y = y_0 y_1 y_2 \dots$ for its words. For concreteness, let us now fix our attention on the change that register B undergoes during the second iteration ($i = 2$) of the algorithm. This change in B is manifest (apart from $S[0]$ and $S[5]$) in y_0 and y_4. In particular, it is easy to verify by tracing through the definition of WEAK that

$$\Big(\Big(y_0 \oplus S[0] \;\oplus\; T[P_1] \;\oplus\; (y_3 \oplus S[3]) \rangle\!\rangle\!\rangle\, 9\Big) \rangle\!\rangle\!\rangle\, 9 \;\oplus\; T[P_5]\Big) \rangle\!\rangle\!\rangle\, 9 \;=\; y_4 \oplus S[4]$$

for some $P_1, P_5 \in \{0, \dots, 511\}$. Distribute $\rangle\!\rangle\!\rangle$ over $\oplus$ and collect up constants and we get that

$$y_4 \;\oplus\; (y_0 \rangle\!\rangle\!\rangle\, 18) \;\oplus\; (y_3 \rangle\!\rangle\!\rangle\, 27) \;\oplus\; c \;=\; (T[P_1] \rangle\!\rangle\!\rangle\, 18) \;\oplus\; (T[P_5] \rangle\!\rangle\!\rangle\, 9)$$

for some constant c. In other words, up to some constant c the adversary can directly "see" in the y's the XOR of a shifted version of pairs of words of T.

To distinguish the output of WEAK from truly random data, simply compute the value of $y_4 \oplus (y_0 \rangle\!\rangle\!\rangle\, 18) \oplus (y_3 \rangle\!\rangle\!\rangle\, 27)$ for each output word y which is seen. If the strings are pseudorandom then this word will take on only 2^{18} possible values, not 2^{32}. From the birthday problem we will be able to make a good prediction of random/pseudorandom using about 2^9 strings y, just by guessing pseudorandom if we see a collision in the $(y_4 \oplus (y_0 \rangle\!\rangle\!\rangle\, 18) \oplus (y_3 \rangle\!\rangle\!\rangle\, 27))$-values in a sample of this size.

Sorting on bits of y_j. Let us go a bit further with the above attack. We witness

$$y_4 \;\oplus\; (y_0 \rangle\!\rangle\!\rangle\, 18) \;\oplus\; (y_3 \rangle\!\rangle\!\rangle\, 27) \;\oplus\; c \;=\; (T[P_1] \rangle\!\rangle\!\rangle\, 18) \;\oplus\; (T[P_5] \rangle\!\rangle\!\rangle\, 9)$$

where P_1 is the offset into T which is the value of P determined in line 1, and P_5 is the offset into T which is the value of P determined in line 5. The thing to notice is that we can tell when two strings y and y' have corresponding P_1 and P_1' which agree. Simply sort the y-values into 512 buckets, depending on the value of the last 9 bits of y_3. All the strings in a given bucket receive the same P_1 value. Thus for the strings y of a given bucket

$$y_4 \;\oplus\; (y_0 \rangle\!\rangle\!\rangle\, 18) \;\oplus\; (y_3 \rangle\!\rangle\!\rangle\, 27)$$

assumes only 512 different values, and these values, apart from a shift, are the entries of T. This forms the basis of a way to reconstruct T.

Guess and verify a correlation between i and $T[i]$. This next attack is based on the fact that because T is small and "randomly-generated" it is not unlikely that there will be substantial correlations between some bit (or small set of bits) of i and some particular bit of $T[i]$. For example, although the least significant bit of i is expected to agree with the 9-th bit of $T[i]$ on 256 out of 512 words, the standard deviation is 11, so it would not be strange if these two bits agreed 240 times, or 270.

Let us index the bits of a word x by $(x)_1(x)_2\ldots(x)_{32}$. Suppose that the least significant bit (bit 32) of i happens to be correlated to the 9-th (bit 9) of $T[i]$. Suppose too that the most significant bit of i (bit 1) happens to be correlated to the 18-th bit of $T[i]$. As an example, maybe $(i)_{32} = (T[i])_9$ 52% of the time, while $(i)_1 \neq (T[i])_{18}$ 53% of the time. The adversary will be able to spot correlations like this, based on a sample of y-values.

Once again, focus on the net change to a particular register which occurs during a particular iteration. To be concrete, let us see how D changes during iteration $i = 2$. First, in line 3, D is modified by a T-value which depends on C. While we don't know what this C-value is, after C is shifted 9 places to the right and XORed with the modified D-value the net change to bit 9 of D is biased according to the direction of the correlation between the least significant bit of i and $(T[i])_9$—in our example, line 3 preserves bit 9 52% of the time and complements bit 9 48% of the time (assuming C is uniformly distributed). Next, on line 4, D is shifted 9 places to the right. This moves the bit in question into position $(D)_{18}$. On line 7 register D is XORed with a table value which depends on C. But this value of C is manifest in the output stream after it has been shifted and masked by the constant $S[5]$. Thus if the 18-th bit of $T[i]$ is correlated with the most significant bit of i, the change to bit 18 of D which line 7 causes will be correlated to bit 10 (due to the right shift of C in line 7) of y_5. Finally, in line 8, the bit in question is shifted into position 27. We conclude that if the initial assumption is correct then there will be a statistical correlation between $(y_2)_9 \oplus (y_6)_{27}$ and $(y_5)_{10}$. This observation can form the basis of a statistical test which looks for "oddities" in the table T.

7 Performance

To get a rough sense of the expected performance of SEAL, we count clock cycles relative to an abstract machine model. Assume a two-operand machine with 32-bit words and at least 7 general purpose registers. Assume that in a single clock cycle we can execute a single addition, logical and, logical exclusive or, data movement, or rotate. Then counting instructions reveals that, if we encrypt long strings with SEAL, we spend about 4 clock cycles per byte. Experimental results on a real machine (see below) are in line with such an estimate.

Some of the efficiency of SEAL stems from the fact that its inner-loop uses only 0.75 table lookups per byte of output. By way of comparison, a software DES implementation typically uses 16 table lookups per byte.

Bosselaers has recently provided us with experimental results on the performance of various cryptographic algorithms [2]. We reproduce some of his data in Figure 4, quoting his figures for the ciphers SEAL, RC4, RC5, and DES, as well as the hash function MD5. For each of these algorithms Bosselaers wrote a highly optimized assembly language implementation for the Pentium processor. Performance of the code was then measured on a 90 MHz machine. For all of the algorithms shown, code and data were resident in on-chip cache. The cost of key setup is ignored. The SEAL figures are for encrypting 1024 bytes of data. They assume a little-endian convention for XORing the plaintext with SEAL's output. The last column in the table gives the speed of SEAL divided by the speed of the indicated algorithm.

Algorithm	Mbit/s	Relative speed
SEAL	198	1.0
RC4	110	1.8
RC5-32/12	38.4	5.2
DES	16.9	11.7
MD5	133.1	1.5

Fig. 4. *Timing figures reported by Bosselaers [2]. The platform is a 90 MHz Intel Pentium processor, and the implementations are in optimized assembly language.*

Bosselaers reports that his SEAL implementation uses 3727 clock cycles to encrypt 1024 bytes. This comes to 3.64 cycles/byte, or 198 Mbit/s with a 90 MHz processor. A total of 4230 instructions are executed to produce these 1024 bytes of output (4.13 instructions/byte), but 1227 of these instructions execute concurrently with the remaining 3003.

A straightforward implementation of SEAL in the language "C" runs at 124 Mbit/s on an SGI Indy with a 100 MHz MIPS 4600 Processor (this is a low-end workstation with a RISC CPU). Compilation was under the Gnu compiler gcc (with optimization), and the code computed $\oplus_{n=0}^{n}\mathsf{SEAL}(a, n, L)$ for a fixed value of a, $L = 1024 \cdot 8$, and a large value of M. The cost of key setup was ignored. The experimental regime ignores the performance penalty which will be incurred if the plaintext, ciphertext, or internal tables of SEAL are out of cache.

The experiments above had SEAL produce output of 1024 bytes, which is an advantageous value for the cipher. When SEAL must produce fewer bytes of output a larger fraction of time is spent on Initialize. For the "C" code mentioned above, producing 512 bytes was 3% slower (per byte) than producing 1024 bytes. Producing 128 bytes was 17% slower. Producing output just more than a multiple of 1024 bytes is also a sub-optimal case for SEAL performance, since little benefit is made of the final call to Initialize.

Key-setup in SEAL has a cost comparable to computing SHA-1 on about 13 KBytes of data; this is estimated to be 2.5–5 msec on a 90 MHz Pentium [3]. In the design of SEAL no attention was paid to minimizing key-setup time. If this is at issue in a target application for SEAL one should select a different method for generating SEAL's tables (e.g., using RC4 or RC5 [13, 14]), or abandon the use of SEAL entirely.

Roe [15] did timing studies of "C" implementations of various cryptographic algorithms, including SEAL 1.0. He used a SUN Sparc and a DEC Alpha. In his experiments on a Sun Sparc, SEAL 1.0 ran 5.4, 11.7, 56.6, and 2.3 times faster than RC4, RC5-32/12, DES, and MD5, respectively. In his experiments on a DEC Alpha, SEAL 1.0 ran 7.6, 15.2, 62.9, and 1.95 times faster than RC4, RC5-32/12, DES, and MD5, respectively. The data indicates a greater speed advantage for SEAL 1.0 than does the data reported by [2]. Probably Roe's "C" code was not uniformly optimized for all of the algorithms.

8 Concluding Remarks

It should be emphasized that using SEAL in the expected way does nothing to provide for data authenticity. Many applications which require data privacy also require data authenticity. Such applications should accompany SEAL-encrypted data by a message authentication code (MAC). Techniques for fast MAC generation are an active area of research.

SEAL is endian-neutral, and yet an endian convention is needed to interoperably encrypt using SEAL. One possibility is to allow encryption with either endian convention, but to include information in SEAL-encrypted ciphertext which unambiguously indicates the endian convention employed.

It is easy to modify SEAL to get a cipher optimized for 64-bit architectures. The tables would be twice as wide and Initialize would be slightly changed. SEAL has the unusual attribute that doubling the word size, and making natural changes in the cipher's definition, would nearly double the cipher's speed. It is unclear whether security would be impacted by the longer word length.

For purposes of possible export approval in various countries, an intentionally weakened version of SEAL can easily be obtained simply by modifying the key generation process. For example, instead of mapping variable-length key a' to underlying 160-bit SEAL key a according to $a = \text{SHA-1}(a')$, one could instead select $a = \text{SHA-1}(\texttt{MASK} \wedge \text{SHA-1}(a'))$, where MASK is a fixed 160-bit mask whose Hamming weight can be adjusted to adjust the security of the cipher.

One thing that the present paper has helped to bring out is the usefulness of designing encryption primitives to be PRFs instead of block ciphers or stream ciphers. A PRF may be easier to use than a stream cipher (because there are no synchronization requirements beyond communicating the index) and easier to make software-efficient than a block cipher.

Acknowledgments

Special thanks to the two anonymous referees for their fine comments and suggestions, and to Antoon Bosselaers for his comments and timely performance data (and for his permission to report some of that data here). Thanks also to Uri Blumenthal and Mike Roe for their performance information.

This work was supported in part by NSF CCR-9624560.

References

1. E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard,* Springer-Verlag, 1993.

2. A. Bosselaers, personal communications, September 1997. Article to appear.
3. A. Bosselaers, R. Govaerts and J. Vandewalle, Fast hashing on the Pentium, *Advances in Cryptology — CRYPTO '96,* Lecture Notes in Computer Science, Vol. 1109, Springer-Verlag, 1996, pp. 298–312.
4. O. Goldreich, S. Goldwasser, and S. Micali, How to construct random functions, *Journal of the ACM,* Vol. 33, No. 4, 1986, pp. 210–217.
5. H. Handschuh and H. Gilbert, χ^2 cryptanalysis of the SEAL encryption algorithm, *Fast Software Encryption*, Lecture Notes in Computer Science, Vol. 1267, Springer-Verlag, 1997, pp. 1–12.
6. G. Marsaglia, The Marsaglia random number CDROM with the DIEHARD battery of tests of randomness. Distributed by the author (geo@stat.fsu.edu) from Florida State University, 1996.
7. A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.
8. R. Merkle, Fast software encryption functions, *Advances in Cryptology — CRYPTO '90,* Lecture Notes in Computer Science, Vol. 537, Springer-Verlag, 1990, pp. 476–501.
9. National Bureau of Standards, Federal Information Processing Standards Publication 46, Data encryption standard. January 1977.
10. National Institute of Standards, U.S. Department of Commerce, FIPS Publication 180-1, Secure hash standard. April 17, 1995 (supersedes FIPS PUB 180).
11. A. Pfitzmann and R. Aßmann, Efficient software implementation of (generalized) DES, *SECURICOM 90: 8-th Worldwide Conference on Computer and Communications Security and Protection,* March 1990.
12. R. Rivest, The MD5 message digest algorithm, RFC 1321 (Internet Request for Comments), April 1992.
13. R. Rivest, unpublished work. (A description of RC4 appears in B. Schneier, *Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, Inc., 1996.)
14. R. Rivest, The RC5 encryption algorithm, *Fast Software Encryption*, Lecture Notes in Computer Science, Vol. 1008, Springer-Verlag, 1995, pp. 86–96.
15. M. Roe, Performance of block ciphers and hash functions — one year later, *Fast Software Encryption*, Lecture Notes in Computer Science, Vol. 809, Springer-Verlag, 1994, pp. 359–362.
16. P. Rogaway and D. Coppersmith, A software-optimized encryption algorithm, *Fast Software Encryption*, Lecture Notes in Computer Science, Vol. 809, Springer-Verlag, 1994, pp. 56–63. (Earlier version of this paper.)
17. B. Schneier, Description of a new variable-length key, 64-bit block cipher (Blowfish), *Fast Software Encryption*, Lecture Notes in Computer Science, Vol. 809, Springer-Verlag, 1994, pp. 191–204.
18. A. Shimizu and S. Miyaguchi, Fast data encryption algorithm FEAL, *Advances in Cryptology — Eurocrypt '87,* Lecture Notes in Computer Science, Vol. 304, Springer-Verlag, 1987.
19. D. Wheeler, A bulk data encryption algorithm, *Fast Software Encryption*, Lecture Notes in Computer Science, Vol. 809, Springer-Verlag, 1994, pp. 127–134.

Appendix A. The Table-Generation Function

We specify $G_a(i)$ for 160-bit string a and integer $0 \leq i < 2^{32}$. 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 [10]; the definition is repeated here only for ease of reference.

First we make the following definitions. For $0 \leq t \leq 19$, set $K_t = $ 0x5a827999 and $f_t(B,C,D) = (B \,\&\, C) \vee (\overline{B} \,\&\, D)$. For $20 \leq t \leq 39$, set $K_t = $ 0x6ed9eba1 and $f_t(B,C,D) = B \oplus C \oplus D$. For $40 \leq t \leq 59$, set $K_t = $ 0x8f1bbcdc and $f_t(B,C,D) = (B \,\&\, C) \vee (B \,\&\, D) \vee (C \,\&\, D)$. For $60 \leq t \leq 79$, set $K_t = $ 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 $W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1$.

c. Let $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$.

d. For $t = 0$ to 79 do
   $\text{TEMP} = A \lll 5 + f_t(B,C,D) + E + W_t + K_t$
   $E = D$; $D = C$; $C = B \lll 30$; $B = A$; $A = \text{TEMP}$;

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

This appendix provides adequate data to verify a correct implementation of SEAL 3.0. 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 $R[0], R[1], \ldots, R[15]$:

```
5021758d ce577c11 fa5bd5dd 366d1b93 182cff72 ac06d7c6 2683ead8 fabe3573
82a10c96 48c483bd ca92285c 71fe84c0 bd76b700 6fdcc20c 8dada151 4506dd64
```

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

```
92b404e5 56588ced 6c1acd4e bf053f68 09f73a93 cd5f176a b863f14e 2b014a2f
4407e646 38665610 222d2f91 4d941a21 aea77ffb 96060a3b 4682af15 13bb3680
........ ........ ........ ........ ........ ........ ........ ........
54e3afcd 301e1c8f 3af3a4bf 021e4080 2a677d95 405c7db0 338e4b1e 19ccf158
```

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

```
907c1e3d ce71ef0a 48f559ef 2b7ab8bc 4557f4b8 033e9b05 4fde0efa 1a845f94
38512c3b d4b44591 53765dce 469efa02 61bea00e a45d6b7d c425744e 53f790ee
........ ........ ........ ........ ........ ........ ........ ........
63d47217 741f96cc bd7dea87 fd036d87 53aa3013 ec60e282 1eaef8f9 0b5a0949
```

Let $n =$ `013577af`. Then $y = \mathsf{SEAL}(a, n, L)$ consists of $y[0] \parallel y[1] \parallel \cdots \parallel y[1023]$:

```
37a00595 9b84c49c a4be1e05 0673530f 5fb097fd f6a13fbd 6c2cdecd 81fdee7c
2abdc3e7 64209aff 00a12283 ef675085 c1634b53 289059e6 a7ab5ed9 480c01eb
........ ........ ........ ........ ........ ........ ........ ........
585a2905 f0496ba5 8eb3d740 efa54b66 4d1a6134 fed9fede 636504aa 691e08e4
```

The XOR of the 1024 words of y is `0x3e0fe99f`.