Scream: a software-efficient stream cipher

Shai Halevi, Don Coppersmith, and Charanjit Jutla

IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA,
{shaih,copper,csjutla}@watson.ibm.com

Abstract. We report on the design of Scream, a new software-efficient stream cipher, which was designed to be a "more secure SEAL". Following SEAL, the design of Scream resembles in many ways a block-cipher design. The new cipher is roughly as fast as SEAL, but we believe that it offers a significantly higher security level. In the process of designing this cipher, we re-visit the SEAL design paradigm, exhibiting some tradeoffs and limitations.

Key words: Stream ciphers, Block ciphers, Round functions, SEAL.

1 Introduction

A stream cipher (or pseudorandom generator) is an algorithm that takes a short random string, and expands it into a much longer string, that still "looks random" to adversaries with limited resources. The short input string is called the seed (or key) of the cipher, and the long output string is called the output stream (or key-stream). Stream ciphers can be used for shared-key encryption, by using the output stream as a one-time-pad. In this work we aim to design a secure stream cipher that has very fast implementations in software.

1.1 A more secure SEAL

The starting point of our work was the SEAL cipher. SEAL was designed in 1992 by Rogaway and Coppersmith [5], specifically for the purpose of obtaining a software efficient stream cipher. Nearly ten years after it was designed, SEAL is still the fastest steam cipher for software implementations on contemporary PC's, with "C" implementations running at 5 cycle/byte on common PC's (and 3.5 cycle/byte on some RISC workstations).

The design of SEAL shares many similarities with the design of common block ciphers. It is built around a repeating *round function*, which provides the "cryptographic strength" of the cipher. Roughly speaking, the main body of SEAL keeps a state which is made of three parts: an *evolving state*, some *round keys*, and a *mask table*. The output stream is generated in steps (or rounds). In each step, the round function is applied to the evolving state, using the round keys. The new evolving state is then masked by some of the entries in the mask table and this value is output as a part of the stream. The mask table is fixed, and some of the round keys are be changed every so often (but not every step).[1]

In terms of security, SEAL is somewhat of a mixed story. SEAL is designed to generate up to 2^{48} bytes of output per seed. In 1997, Handschuh and Gilbert showed, however, that the output stream can be distinguished from random after seeing roughly 2^{34} bytes of output [4]. SEAL was slightly modified after that attack, and the resulting algorithm is known as SEAL 3.0. Recently, Fluhrer described an attack on SEAL 3.0, that can distinguish the output stream from random after about 2^{44} output bytes [3]. Hence, it seems prudent to avoid using the same seed for more than about 2^{40} bytes of output.

The goal of the current work was to come up with a "more secure SEAL". As part of that, we studied the advantages, drawbacks, and tradeoffs of this style of design. More specifically, we tried to understand what makes a "good round function" for a stream cipher, and to what extent a "good round function" for a block cipher is also good as the basis for a stream cipher. We also studied the interaction between the properties of the round function and other parts of the cipher. Our design goals for the cipher were as follows:

- Higher security than SEAL: It should be possible to use the same seed for 2^{64} bytes of output. More precisely, an attacker that sees a total of 2^{64} bytes of output (possibly, using several IV's of its choice), would be forced to spend an infeasible amount of time (or space) in order to distinguish the cipher from a truly random function. A reasonable measure of "infeasibility" is, say, 2^{80} space and 2^{96} time, so we tried to get the security of the cipher comfortably above these values.[2]
- Comparable speed to SEAL, i.e., about 5 cycles per byte on common PC's.
- We want to allow a full 128-bit input nonces (vs. 32-bit nonce in SEAL).
- Other, secondary, goals were to use smaller tables (SEAL uses 4KB of secret tables), get faster initialization (SEAL needs about 200 applications of SHA to initialize the tables), and maybe make the cipher more amenable to implementation in other environments (e.g., hardware, smartcard, etc.) We also tried to make the cipher fast on both 32-bit and 64-bit architectures.

1.2 The end result(s)

In this report we describe three variants of our cipher. The first variant, which we call Scream-0, should perhaps be viewed as a "toy cipher". Although it may be secure enough for some applications, it does not live up to our security goals. In the full version of this report we describe a "low-diffusion attack" that works in time 2^{79} and space 2^{50}, and distinguishes Scream-0 from random after seeing about 2^{44} bytes of the output stream.

---

[1] In SEAL, the evolving state is the words A, B, C, D, the round keys consists of the table T and the n_i's, and the mask table is S.

[2] This security level is arguably lower than, say, AES. This seems to be the price that one has to pay for the increased speed. We note that the "obvious solution" of using Rijndael with less rounds, fails to achieve the desired security/speed tradeoff.

We then describe Scream, which is the same as Scream-0, except that it replaces the fixed S-boxes of Scream-0 by key-dependent S-boxes. Scream has very fast software implementations, but to get this speed one has to use secret tables roughly as large as those of SEAL (mainly, in order to store the S-boxes). On our Pentium-III machine, an optimized "C" implementation of Scream runs at 4.9 cycle/byte, slightly faster than SEAL. On a 32-bit PowerPC, the same implementation runs at 3.4 cycle/byte, again slightly faster than SEAL. This optimized implementation of Scream uses about 2.5 KB of secret tables. Scream also offers some space/time tradeoffs. (In principle, one could implement Scream with less than 400 bytes of memory, but using so little space would imply a slowdown of at least two orders of magnitude, compared to the speed-optimized implementation.) In terms of security, if the attacker is limited to only 2^{64} bytes of text, we do not know of any attack that is faster than exhaustively searching for the 128-bit key. On the other hand, we believe that it it possible to devise a linear attack to distinguish Scream from random, with maybe 2^{80} bytes of text.

At the end of this report we describe another variant, called Scream-F (for Fixed S-box), that does not use secret S-boxes, but is slower than Scream (and also somewhat "less elegant"). An optimized "C" implementation of Scream-F runs at 5.6 cycle/byte on our Pentium-III, which is 12% slower than SEAL. On our PowerPC, this implementation runs at 3.8 cycle/byte, 10% slower than SEAL. This implementation of Scream-F uses 560 bytes of secret state. We believe that the security of Scream-F is roughly equivalent to that of Scream.

1.3 Organization

In Section 2 below we first describe Scream-0 and then Scream. In Section 3 we discuss implementation issues and provide some performance measurements. In Section 4 we briefly discuss the cryptanalysis of Scream-0. (A more detailed analysis can be found in the full version.) Finally, in Section 5, we describe the cipher Scream-F. In the appendix we give the constants that are used in Scream, and also provide some "test vectors".

2 The design of Scream

We begin with the description of Scream-0. As with SEAL, this cipher too is built around a "round function" that provides the cryptographic strength. Early in our design, we tried to use an "off the shelf" round function as the basis for the new cipher. Specifically, we considered using the Rijndael round function [2], which forms the basis of the new AES. However, as we discuss in the full paper, the "wide trail strategy" that underlies the design of the Rijndael round function is not a very good match for this type of design. We therefore designed our own round function.

At the heart of our round function is a scaled-down version of the Rijndael function, that operates on 64-bit blocks. The input block is viewed as a 2×4 matrix of bytes. First, each byte is sent through an S-box, $S[\cdot]$, then the second row in the matrix is shifted cyclically by one byte to the right, and finally each column is multiplied by a fixed 2×2 invertible matrix M. Below we call this function the “half round function”, and denote it by $G_{S,M}(x)$. A pictorial description of $G_{S,M}$ can be found in Figure 1.

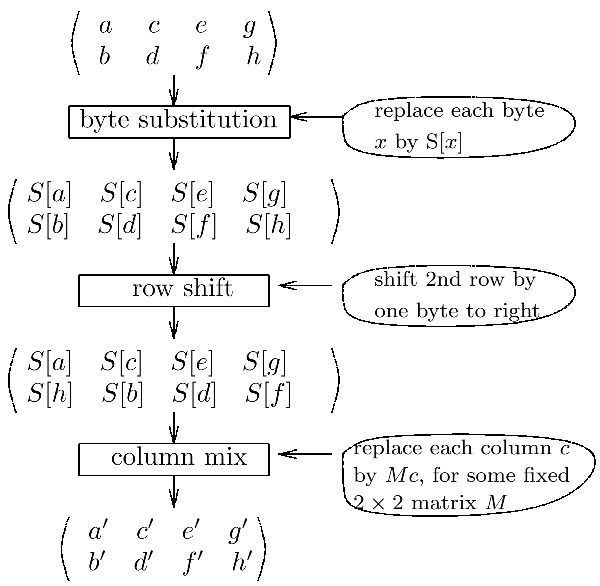

Fig. 1. The “half round” function $G_{S,M}$

Our round function, denoted $F(x)$, uses two different instances of the “half-round” function, G_{S_1,M_1} and G_{S_2,M_2}, where S_1, S_2 are two different S-boxes, and M_1, M_2 are two different matrices. The S-boxes S_1, S_2 in Scream-0 are derived from the Rijndael S-box, by setting $S_1[x] = S[x]$, and $S_2[x] = S[x \oplus 00010101]$, where $S[\cdot]$ is the Rijndael S-box. The constant 00010101 (decimal 21) was chosen so that S_2 will not have a fixed-point or an inverse fixed-point.[3] The matrices M_1, M_2 were chosen so that they are invertible, and so that neither of M_1, M_2 and $M_2^{-1} M_1$ contains any zeros. Specifically, we use

$$M_1 = \begin{pmatrix} 1 & x \\ x & 1 \end{pmatrix} \qquad M_2 = \begin{pmatrix} 1 & x+1 \\ x+1 & 1 \end{pmatrix}$$

where $1, x, x+1$ are elements of the field $GF(2^8)$, which is represented as $\mathbb{Z}_2[x]/(x^8 + x^7 + x^6 + x + 1)$.

The function F is a mix of a Feistel ladder and an SP-network. A pseudocode of F is provided below, and a pictorial description can be found in Figure 2.

[3] An inverse fixed-point is some x such that $S[x] = \bar{x}$.

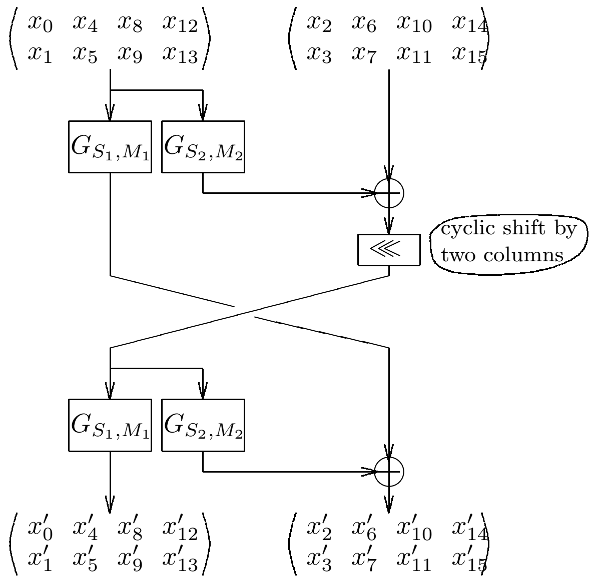

Fig. 2. The round function, F

Function $F(x)$:

1. Partition x into two 2×4 matrices
$$A := \begin{pmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \end{pmatrix} \qquad B := \begin{pmatrix} x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{pmatrix}$$
2. $B := B \oplus G_{S_2,M_2}(A)$ // use A to modify A, B
3. $A := G_{S_1,M_1}(A)$
4. $B := \begin{pmatrix} B_{0,2} & B_{0,3} & B_{0,0} & B_{0,1} \\ B_{1,2} & B_{1,3} & B_{1,0} & B_{1,1} \end{pmatrix}$ // rotate B by two columns
5. Swap $A \leftrightarrow B$
6. $B := B \oplus G_{S_2,M_2}(A)$ // use A to modify A, B
7. $A := G_{S_1,M_1}(A)$
8. Collect the 16 bytes in A, B back into x
$$x' := (A_{0,0}\ A_{1,0}\ B_{0,0}\ B_{1,0}\ A_{0,1}\ A_{1,1}\ B_{0,1}\ B_{1,1}\ A_{0,2}\ A_{1,2}\ B_{0,2}\ B_{1,2}\ A_{0,3}\ A_{1,3}\ B_{0,3}\ B_{1,3})$$

The main loop of Scream-0. As with SEAL, the cipher Scream-0 maintains a state that consists of the "evolving state" x, some round keys y, z, and a "mask table" W. In Scream-0, x, y and z are 16-byte blocks, and the table W consists of 16 blocks, each of 16 bytes. In step i of Scream-0, the evolving state is modified by setting $x := F(x \oplus y) \oplus z$, and we then output $x \oplus W[i \bmod 16]$.

In Scream-0, both the mask table and the round keys are modified, albeit slowly, throughout the computation. Specifically, after every pass through the mask table (i.e., every 16 steps), we modify y, z and one entry in W, by passing them through the F function. The entries of W are modified in order: after the j'th pass through the table we modify the entry $W[j \bmod 16]$. Moreover, instead of keeping both y, z completely fixed for 16 rounds, we rotate y by a few bytes after each use. The rotation amounts were chosen so that the rotation would be "almost for free" on 32-bit and 64-bit machines. This simple measure provides some protection against "low-diffusion attacks" and linear analysis. A pseudocode of the body of Scream-0 is described in Figure 3.

```
The main loop of Scream:
State: x, y, z – three 16-byte blocks
       W       – a table of 16 16-byte blocks
       i_w     – an index into W (initially i_w = 0)
repeat (until you get enough output bytes)
    for i = 0 to 15                      // generate the next 16 output blocks
        x := F(x ⊕ y)                                // modify the "evolving state" x
        x := x ⊕ z
        output x ⊕ W[i mod 16]
        if i = 0 or 2 mod 4                          // rotate y
            rotate y by 8 bytes, y := y_{8..15,0..7}
        else if i = 1 mod 4
            rotate each half of y by 4 bytes, y := y_{4..7,0..3,12..15,8..11}
        else if i < 15                               // no point in rotating when i = 15
            rotate each half of y by three bytes to the right, y := y_{5..7,0..4,13..15,8..12}
        end-if
    end-for
    y := F(y ⊕ z)                        // modify y, z, and W[i_w]
    z := F(z ⊕ y)
    W[i_w] := F(W[i_w])
    i_w := i_w + 1 mod 16
end-repeat
```

Fig. 3. The main body of Scream and Scream-0

Key- and nonce-setup. The key- and nonce-setup procedures of Scream-0 are quite straightforward: We just use the round function F to derive all the quantities that we need. The key-setup routine fills the table W with some initial values. These values are later modified during the nonce-setup routine, and they also double as the equivalent of a "key schedule" for the nonce-setup routine. A pseudocode for these two routines is provided in Figures 4 and 5.

<u>Key-setup:</u>

Input:	seed	– a 16-byte block
State:	a, b	– temporary variables, each a 16-byte block
Output:	$W0$	– a table of sixteen 16-byte blocks

0a. set $S_1[x] := S[x]$ for all x // $S[\cdot]$ is the Rijndael S-box
0b. set $S_2[x] := S_1[x \oplus 00010101]$ for all x

1. $a :=$ seed
2. $b := F(a \oplus pi)$ // pi is a constants: the first 16 bytes in the binary expansion of π
3. for $i = 0$ to 15
4. $\quad a := F^4(a) \oplus b$ // four applications of the function F
5. $\quad W0[i] := a$
6. end-for

Fig. 4. The key-setup of Scream-0

<u>Nonce-setup:</u>

Input:	nonce	– a 16-byte block
State:	$W0$	– a table of sixteen 16-byte blocks
	a, b	– temporary variables, each a 16-byte block
Output:	x, y, z	– three 16-byte blocks
	W	– a table of sixteen 16-byte blocks

1. $z := F^2(\text{nonce} \oplus W0[1])$ // two applications of the function F
2. $y := F^2(z \oplus W0[3])$
3. $a := F^2(y \oplus W0[5])$
4. $x := F(a \oplus W0[7])$ // only one application of F
5. $b := x$
6. for $i = 0$ to 7 // set W as a modification of $W0$
7. $\quad b := F(b \oplus W0[2i])$
8. $\quad W[\ 2i\ ] \ := W0[\ 2i\ ] \oplus a$
9. $\quad W[2i+1] := W0[2i+1] \oplus b$
10. end-for

Fig. 5. The nonce-setup of Scream and Scream-0

2.1 The ciphers Scream

The cipher Scream is the same as Scream-0, except that we derive the S-boxes $S_1[\cdot], S_2[\cdot]$ from the Rijndael S-box $S[\cdot]$ in a key-dependent fashion. We replace line 0a in Figure 4 by the following

$$\text{0a. set } S_1[x] := S[\dots S[S[x + \text{seed}_0] + \text{seed}_1] \dots + \text{seed}_{15}] \text{ for all } x$$

(Notice that $+$ denotes integer addition mod 256, rather then exclusive-or.) In terms of speed (in software), Scream-S is just as fast as Scream-0, except for the key-setup. However, it has a much larger secret state (a speed-optimized software implementation of Scream-S uses additional 2Kbyte of secret tables). We note that we still have $S_2[x] = S_1[x \oplus 00010101]$, so a space-efficient implementation need only store S_1.

3 Implementation and performance

Software implementation of the F function. A fast software implementation of the F function uses tricks similar to Rijndael: Namely, we can implement the two "half round" functions G_{S_1,M_1}, G_{S_2,M_2} together, using just eight lookup operations into two tables, each consisting of 256 four-byte words. Let the eight-byte input to G_{S_1,M_1}, G_{S_2,M_2} be denoted $(x_0, x_1,\ x_4, x_5,\ x_8, x_9,\ x_{12}, x_{13})$, the output of G_{S_1,M_1} be denoted $(u_0, u_1,\ u_4, u_5,\ u_8, u_9,\ u_{12}, u_{13})$, and the output of G_{S_2,M_2} be denoted $(u_2, u_3,\ u_6, u_7,\ u_{10}, u_{11},\ u_{14}, u_{15})$. Then we can write:

$$\begin{aligned}
u_0 &= M_1(0,0) \cdot S1[x_0] \oplus M_1(0,1) \cdot S1[x_{13}] \\
u_1 &= M_1(1,0) \cdot S1[x_0] \oplus M_1(1,1) \cdot S1[x_{13}] \\
u_2 &= M_2(0,0) \cdot S2[x_0] \oplus M_2(0,1) \cdot S2[x_{13}] \\
u_3 &= M_2(1,0) \cdot S2[x_0] \oplus M_2(1,1) \cdot S2[x_{13}]
\end{aligned}$$

(where $M(i,j)$ is the entry in row i, column j of matrix M, indexing starts from zero). Similar expressions can be written for the other bytes of u. Therefore, if we set the tables T_0, T_1 as

$$T_0(x) = \left\langle M_1(0,0) \cdot S1[x] \;\middle|\; M_1(1,0) \cdot S1[x] \;\middle|\; M_2(0,0) \cdot S2[x] \;\middle|\; M_2(1,0) \cdot S2[x] \right\rangle$$

$$T_1(x) = \left\langle M_1(0,1) \cdot S1[x] \;\middle|\; M_1(1,1) \cdot S1[x] \;\middle|\; M_2(0,1) \cdot S2[x] \;\middle|\; M_2(1,1) \cdot S2[x] \right\rangle$$

Then we can compute $u_{0..3} := T_0[x_0] \oplus T_1[x_{13}]$, $u_{4..7} := T_0[x_4] \oplus T_1[x_1]$, $u_{8..11} := T_0[x_8] \oplus T_1[x_5]$, and $u_{12..15} := T_0[x_{12}] \oplus T_1[x_9]$. A "reasonably optimized" implementation of the round function F (on a 32-bit machine) may work as follows:

<u>Function $F(x_0, x_1, x_2, x_3)$:</u> // each x_i is a four-byte word
Temporary storage: u_0, u_1, u_2, u_3, each a four-byte word
1. $u_0 := T_0[\text{byte0}(x_0)] \oplus T_1[\text{byte1}(x_3)]$ // first "half round"
2. $u_1 := T_0[\text{byte0}(x_1)] \oplus T_1[\text{byte1}(x_0)]$

3. $u_2 := T_0[\text{byte0}(x_2)] \oplus T_1[\text{byte1}(x_1)]$
4. $u_3 := T_0[\text{byte0}(x_3)] \oplus T_1[\text{byte1}(x_2)]$
5. $[\text{byte2}(u_0) \mid \text{byte3}(u_0)] \;:=\; [\text{byte2}(u_0) \mid \text{byte3}(u_0)] \oplus [\text{byte2}(x_0) \mid \text{byte3}(x_0)]$
6. $[\text{byte2}(u_1) \mid \text{byte3}(u_1)] \;:=\; [\text{byte2}(u_1) \mid \text{byte3}(u_1)] \oplus [\text{byte2}(x_1) \mid \text{byte3}(x_1)]$
7. $[\text{byte2}(u_2) \mid \text{byte3}(u_2)] \;:=\; [\text{byte2}(u_2) \mid \text{byte3}(u_2)] \oplus [\text{byte2}(x_2) \mid \text{byte3}(x_2)]$
8. $[\text{byte2}(u_3) \mid \text{byte3}(u_3)] \;:=\; [\text{byte2}(u_3) \mid \text{byte3}(u_3)] \oplus [\text{byte2}(x_3) \mid \text{byte3}(x_3)]$

9. $u_0 := u_0 \lll 2$ bytes // swap the two halves
10. $u_1 := u_1 \lll 2$ bytes
11. $u_2 := u_2 \lll 2$ bytes
12. $u_3 := u_3 \lll 2$ bytes

13. $x_0 := T_0[\text{byte0}(u_2)] \oplus T_1[\text{byte1}(u_2)]$ // second "half round"
14. $x_1 := T_0[\text{byte0}(u_3)] \oplus T_1[\text{byte1}(u_3)]$
15. $x_2 := T_0[\text{byte0}(u_0)] \oplus T_1[\text{byte1}(u_0)]$
16. $x_3 := T_0[\text{byte0}(u_1)] \oplus T_1[\text{byte1}(u_1)]$
17. $[\text{byte2}(x_0) \mid \text{byte3}(x_0)] \;:=\; [\text{byte2}(x_0) \mid \text{byte3}(x_0)] \oplus [\text{byte2}(u_0) \mid \text{byte3}(u_0)]$
18. $[\text{byte2}(x_1) \mid \text{byte3}(x_1)] \;:=\; [\text{byte2}(x_1) \mid \text{byte3}(x_1)] \oplus [\text{byte2}(u_1) \mid \text{byte3}(u_1)]$
19. $[\text{byte2}(x_2) \mid \text{byte3}(x_2)] \;:=\; [\text{byte2}(x_2) \mid \text{byte3}(x_2)] \oplus [\text{byte2}(u_2) \mid \text{byte3}(u_2)]$
20. $[\text{byte2}(x_3) \mid \text{byte3}(x_3)] \;:=\; [\text{byte2}(x_3) \mid \text{byte3}(x_3)] \oplus [\text{byte2}(u_3) \mid \text{byte3}(u_3)]$

21. output (x_0, x_1, x_2, x_3)

We note the need for explicit swapping of the two halves above (lines 9-12). The reason for that is that the tables T_0, T_1 are arranged so that the part corresponding to G_{S_1,M_1} is in the first two bytes of each entry, and the part of G_{S_2,M_2} is in the last two bytes. The code above can be optimized further, combining the rotation in these lines with the masking, which is implicit in lines 5-8, 17-20. Hence, the rotation becomes essentially "for free".

This structure provides a space/time tradeoff similar to Rijndael: Since the matrices M_1, M_2 are symmetric, one can obtain $T_2(x)$ from $T_1(x)$ using a few shift operations. Hence, it is possible to store only one table, at the expense of some slowdown in performance. This tradeoff is particularly important for Scream, where the tables T_0, T_1 are key-dependent.

The nonce-setup routine. The nonce-setup routine was designed so that the first output block can be computed as soon as possible. Although all the entries of the table W have to be modified during the nonce-setup, an application that does not use all of them can modify only as many as it needs. Hence an application that only outputs a few blocks per input nonce, does not have to complete the entire nonce-setup. Alternatively, an application can execute the nonce-setup together with the first "chunk" of 16 steps, modifying each mask of W just before this mask is needed.

Performance in software. We tested the software performance of Scream and Scream-F on two platforms, both with word-length of 32 bits: One platform is an IBM PC 300PL, with a 550MHz Pentium-III processor, running Linux and using the gcc compiler, version 3.0.3. The other platform is an RS/6000 43P-150 workstation, with a 375MHz 304e PowerPC processor, running AIX 4.3.3 and using the IBM C compiler (xlc) version 3.6.6. On both platforms, we measured peak throughput, and also timed the key-setup and nonce-setup routines. To measure peak throughput, we timed a procedure that produces 256MB of output (all with the same key and nonce). Specifically, the procedure makes one million calls to a function that outputs the next 256 bytes of the cipher. To eliminate the effect of cache misses, we used the same output buffer in all the calls. We list our test results in the table below.

Platform	Operation	Scream-F	Scream	SEAL
Pentium-III	throughput	5.6 cycle/byte	4.9 cycle/byte	5.0 cycle/byte
550 MHz	key-setup	3190 cycles	27500 cycles	
Linux, gcc	nonce-setup	1276 cycles	1276 cycles	
604e PowerPC	throughput	3.8 cycle/byte	3.4 cycle/byte	3.45 cycle/byte
375 MHz	key-setup	1950 cycles	16875 cycles	
AIX, xlc	nonce-setup	670 cycles	670 cycles	

Implementation in different environments. Being based on a Rijndael-like round function, Scream is amenable for implementations in many different environments. In particular, it should be quite easy to implement it in hardware, and the area/speed tradeoff in such implementation may be similar to Rijndael (except that Scream needs more memory for the mask table). Also, it should be quite straightforward to implement it for 8- and 16-bit processors (again, as long as the architecture has enough memory to store the internal state). Scream is clearly not suited for environments with extremely small memory, but it can be implemented with less than 400 bytes of memory (although such implementation would be quite slow).

4 Security Analysis

Below we examine some possible attacks on Scream-0 and Scream. The discussion below deals mostly with Scream-0. At the end we briefly discuss the effect of Scream's key-dependent S-boxes on these attacks. We examine two types of attacks, one based on linear approximations of the F function, and the other exploits the low diffusion provided by a single application of F. In both attacks, the goal of the attacker is to distinguish the output of the cipher from a truly random stream.[4]

4.1 Linear attacks

It is not hard to see that the F function has linear approximations that approximate only three of the 8-by-8 S-boxes. Since the S-boxes in Scream-0 are based

[4] In a separate paper [1], we show that these two types of attacks can be viewed as two special cases of a generalized distinguishing attack.

on the Rijndael S-box, the best approximation of them has bias 2^{-3}, so we can probably get a linear approximation of the F function with bias 2^{-9}. Namely, there exists a linear function L such that $\Pr_x[L(x, F(x)) = 0] = 1/2 \pm 2^{-9}$. (In the full version of this report we show that bias of 2^{-9} is indeed the best possible.)

To use this approximation, we need to eliminate the linear masking, introduced by the y, z and the $W[i]$'s. Here we use the fact that each one of these masks is used sixteen times before it is modified. For each step of the cipher, the attacker sees a pair $(x \oplus y \oplus W[i],\ F(x) \oplus z \oplus W[i+1])$, where x is random. Applying the function L to this pair, we get the bit

$$\sigma \ = \ L(x, F(x)) \oplus L(y, z) \oplus L(W[i], W[i+1])$$

For simplicity, we ignore for the moment the rotation of the y block after each step. If we add two such σ's that use the same y and z blocks, we get $\tau = \sigma \oplus \sigma' = L(x, F(x)) \oplus L(x', F(x')) \oplus L(W[i], W[i+1]) \oplus L(W[j], W[j+1])$. The last bit does not depend on y, z anymore. We can repeat this process, adding two such τ's that use the same masks, we end up with a bit

$$\mu = \tau \oplus \tau' = L(x, F(x)) \oplus L(x', F(x')) \oplus L(x'', F(x'')) \oplus L(x''', F(x'''))$$

Since $L(x, F(x))$ has bias of 2^{-9}, the bit μ has bias of 2^{-36}, so after seeing about 2^{72} such bits, we can distinguish the cipher from random.

Since each of the masks is used sixteen times before it is modified, we have about $\binom{16}{2}$ choices for the pairs of σ's to add (still ignoring the rotation of y), and about $\binom{16}{2}$ choices for the pairs of τ's to add. Hence, 256 steps of the cipher gives us about $\binom{16}{2}^2 \approx 2^{14}$ bits μ. After seeing roughly $256 \cdot 2^{58} = 2^{66}$ steps of the cipher (i.e., 2^{70} bytes of output), we can to collect the needed 2^{72} samples of μ's to distinguish the cipher from random.

The rotation of y. The rotation of y makes it harder to devise attacks as above. To cancel both the y and the z blocks, one would have to use two different approximations with the same output bit pattern, but where the input bit patterns are rotated accordingly. We do not know if it possible to devise such approximation with bias of 2^{-9}.

The secret S-boxes. The introduction of key-dependent S-boxes in Scream does not significantly alter the analysis from above. Since the S-boxes are key-dependent, an attacker cannot pick "the best approximations" for them, but on the other hand these S-boxes have better approximations than the Rijndael S-box. Thus, the attacker can use a random approximation, and it will likely to be roughly as good as the best approximation for the fixed S-boxes.

4.2 Low-diffusion attacks

A low-diffusion attack exploits the fact that not every byte of $F(x)$ is influenced by every byte of x (and vise versa). For example, there are output bytes that only depend on six input bytes. In fact, in the full version of this report we show that knowing two bytes of x and one byte of (linear combination of bytes in) $F(x)$, we can compute another byte of (linear combination of bytes in) $F(x)$. Namely, we have a (non-degenerate) linear function L with output length of four bytes, so that we can write $L(X, F(x))_3 = g(L(X, F(x))_{0..2})$, where g is an known deterministic function (with three bytes of input and one byte of output).

As for the linear attacks, here too we need to eliminate the linear masking, introduced by the y, z and the $W[i]$'s. This is done in very much the same way. Again, we ignore for now the rotation of the block y. For each step of the cipher the attacker sees the four bytes $L(x \oplus y \oplus W[i], F(x) \oplus z \oplus W[i+1])$. We eliminate the dependence on y, z by adding two such quantities that use the same y, z blocks. This gives a four-byte quantity $L(x, F(x)) \oplus L(x', F(x')) \oplus L(W[i], W[i+1]) \oplus L(W[j], W[j+1])$. Adding two of those with the same i, j, we then obtain the four byte quantity

$$L(x, F(x)) \oplus L(x', F(x')) \oplus L(x'', F(x'')) \oplus L(x''', F(x'''))$$

We can write this last quantity in terms of the function g, as a pair $(r_1 \oplus r_2 \oplus r_3 \oplus r_4,\ g(r_1) \oplus g(r_2) \oplus g(r_3) \oplus g(r_4)\ )$, where the r_i's are three-byte long, and the $g(r_i)$'s are one-byte long. In a separate paper [1], we analyze the statistical properties of such expressions, and calculate the number of samples that needs to be seen to distinguish them from random.

The rotation of y. Again, the rotation of y makes it harder to devise attacks as above. In the full paper we show, however, that we can still use a low-diffusion attack on the F function, in which guessing six bytes of $(x, F(x))$ yields the value of four other bytes. Applying tools from our paper [1] to this relation, we can compute that the amount of output text that is needed to distinguish the cipher from random along the lines above, is merely 2^{44} bytes. However, the procedure for distinguishing is quite expensive. The most efficient way that we know how to use these 2^{44} bytes would require roughly 2^{50} space and 2^{80} time.

The secret S-boxes. At present, we do not know how to extend low-diffusion attacks such as above to deal with secret S-boxes. Although we can still write the same expression $L(X, F(x))_3 = g(L(X, F(x))_{0..2})$, the function g now depends on the key, so it is not known to the attacker. Although it is likely that some variant of these attacks can be devised for this case too, we strongly believe that such variants would require significantly more text than the 2^{64} bytes that we "allow" the attacker to see.

5 The cipher Scream-F

In Scream, we used key-dependent S-boxes to defend against "low-diffusion attacks". A different approach is to keep the S-box fixed, but to add to the main body of the cipher some "key dependent operation" before outputting each block.

This approach was taken in Scream-F, where we added one round of Feistel ladder after the round function, using a key-dependent table. However, since the only key-dependent table that we have is the mask table W, we let W double also as an "S-box". Specifically, we add the following lines 3a-3e between lines 3 and 4 in the main-loop routine from Figure 3.

3a.	view the table W as an array of 64 4-byte words $\hat{W}[0..63]$
3b.	$x_{0..3} := x_{0..3} \oplus \hat{W}[1 + (x_4 \wedge 00111110)]$
3c.	$x_{4..7} := x_{4..7} \oplus \hat{W}[x_8 \wedge 00111110]$
3d.	$x_{8..11} := x_{8..11} \oplus \hat{W}[1 + (x_{12} \wedge 00111110)]$
3e.	$x_{12..15} := x_{12..15} \oplus \hat{W}[x_0 \wedge 00111110]$

We note that the operation $x_i \wedge 00111110$ in these lines returns an even number between 0 and 62, so we only use odd entries of W to modify $x_{0..3}$ and $x_{8..11}$, and even entries to modify $x_{4..7}$ and $x_{12..15}$. The reason is that to form the output block, the words $x_{0..3}, x_{8..11}$ will be masked with even entries of W, and the words $x_{4..7}, x_{12..15}$ will be masked by odd entries. The odd/even indexing is meant to avoid the possibility that these masks cancel with the entries that were used in the Feistel operation.[5]

5.1 Conclusions

We presented Scream, a new stream cipher with the same design style as SEAL. The new cipher is roughly as fast as SEAL, but we believe that it is more secure. It has some practical advantages over SEAL, in flexibility of implementation, and also in the fact that it can take a full 128-bit nonce (vs. 32 bits in SEAL). In the process of designing Scream, we studied the advantages and pitfalls of the SEAL design style. We hope that the experience from this work would be beneficial also for future ciphers that uses this style of design.

Acknowledgments This design grew out of a study group in IBM, T.J. Watson during the summer and fall of 2000. Other than the authors, the study group also included Ran Canetti, Rosario Gennaro, Nick Howgrave-Graham, Tal Rabin and J.R. Rao. The motivation for this work was partly due to the NESSIE "call for cryptographic primitives" (although we missed their deadline by more than one year).

References

1. D. Coppersmith, S. Halevi, and C. Jutla. Cryptanalysis of stream ciphers with linear masking. manuscript, 2002.

[5] It is still possible that two words, say $x_{0..3}$ and $x_{4..7}$, are masked with the same mask, but it seems less harmful.

2. J. Daemen and V. Rijmen. AES proposal: Rijndael. Available on-line from NIST at `http://csrc.nist.gov/encryption/aes/rijndael/`, 1998.
3. S. Fluhrer. Cryptanalysis of the SEAL 3.0 pseudorandom function family. In *Proceedings of the Fast Software Encryption Workshop (FSE'01)*, 2001.
4. H. Handschuh and H. Gilbert. χ^2 cryptanalysis of the SEAL encryption algorithm. In *Proceedings of the 4th Workshop on Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1997.
5. P. Rogaway and D. Coppersmith. A software optimized encryption algorithm. *Journal of Cryptology*, 11(4):273–287, 1998.

A Constants and Test Vectors

```
The Rijndael S-box, S[0..255] = [
  63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76 ca 82 c9 7d fa 59 47 f0
  ad d4 a2 af 9c a4 72 c0 b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
  04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75 09 83 2c 1a 1b 6e 5a a0
  52 3b d6 b3 29 e3 2f 84 53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
  d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8 51 a3 40 8f 92 9d 38 f5
  bc b6 da 21 10 ff f3 d2 cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
  60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db e0 32 3a 0a 49 06 24 5c
  c2 d3 ac 62 91 95 e4 79 e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
  ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a 70 3e b5 66 48 03 f6 0e
  61 35 57 b9 86 c1 1d 9e e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
  8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16 ]

The constant pi (for key-setup)
    pi        = [24 3f 6a 88 85 a3 08 d3 13 19 8a 2e 03 70 73 44]


Test vectors for Scream-S
    *** key-setup test vectors ***
    key       = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
    W0[0]     = [b6 a5 0b bf f3 9b 9e 99 28 b0 35 18 7b 7d 9c 7b]
    W0[15]    = [83 32 53 22 db 10 00 31 49 3a a4 80 3a 41 8c b3]

    *** nonce-setup test vectors ***
    key       = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
    nonce     = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
    X         = [b4 b7 7e 35 6a 24 0c c8 a7 41 b8 c7 d7 29 68 82]
    Y         = [e4 f4 1d 3b fd 07 d4 3c cb df a9 bb 25 df 65 6c]
    Z         = [87 de 72 cd 96 5a 96 24 b4 eb 79 66 57 26 fd f9]
    W[0]      = [66 d4 35 4d 2c 90 5f 0e 7f cc 25 59 43 ba d2 22]
    W[15]     = [a8 0e b6 56 be aa 5d d2 8d ca fe 07 1b f9 9c 7a]

    *** stream test vectors ***
    key       = [01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10]

    nonce     = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
    out[0]    = [74 8c 59 f2 0d 76 9e a8 7a 6d c1 87 46 e6 4a c0]
```

```
    out[1]    = [bd 3b 39 cd 12 18 43 0f 80 fa e0 1b 2e 60 f1 74]
    out[4]    = [15 21 8a 46 fb ee 26 54 98 8d 2b 80 8a 87 f4 5e]
    out[16]   = [cb 32 f4 d6 f7 ce 57 69 e2 a3 ac d8 37 e1 37 82]
    out[1023] = [97 ec 87 f0 a0 6c e7 0b 75 e6 12 25 50 1f 82 e3]

    nonce     = [01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01]
    out[0]    = [47 68 06 37 83 85 99 af d2 8f fb 2e dd fc 9d 2e]
    out[1]    = [7b d3 0b e4 7a a6 3b 5f 4f 5f 05 06 66 17 d5 a2]
    out[4]    = [98 aa 20 75 73 c7 fa fc 1c 4c 27 61 46 14 3c 1d]
    out[16]   = [b3 33 a4 8e 17 50 8e ab b2 68 fb 60 67 56 46 1e]
    out[1023] = [a5 41 b3 37 c6 bd 8a 4b 41 a1 40 5f ea c5 a3 f5]


Test vectors for Scream-F
    *** key-setup test vectors ***
    key       = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
    W0[0]     = [be a0 cd 9a 5d f6 85 59 c0 3f a9 c5 53 fd ad e1]
    W0[15]    = [eb 2e ab 45 26 ee 49 e1 34 db 97 87 62 d1 3b 25]

    *** nonce-setup test vectors ***
    key       = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
    nonce     = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
    X         = [d4 10 c5 bf bd 7b fd 81 37 4e e3 b0 c1 bf 8b a6]
    Y         = [51 a6 7f 38 3d 0d 95 26 bf b5 b0 e8 26 b5 e4 93]
    Z         = [53 50 b7 d6 87 3d df 8c 7f 9b 10 7c e0 92 d0 02]
    W[0]      = [cb ad d5 c2 b0 85 af 77 6c d8 ef ce 7b 36 65 3a]
    W[15]     = [19 14 5e 0a 4d 23 1c d5 f9 6f 85 8a 39 38 81 a1]

    *** stream test vectors ***
    key       = [01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10]

    nonce     = [00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00]
    out[0]    = [39 ec 4a 06 45 4d c3 cd 96 dd ef 0c f0 c2 67 40]
    out[1]    = [a0 ea 56 e7 e3 c8 f5 df 34 ea 35 ee 77 ed da 66]
    out[4]    = [8a c8 93 af 83 ed 0a 53 6b e9 f4 7c b6 6d 21 67]
    out[16]   = [e0 8c fe 31 34 a7 48 ca 14 10 f9 58 50 71 49 20]
    out[1023] = [a4 e2 fc be 0a 47 53 9a 23 e0 79 25 5c be ea e7]

    nonce     = [01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01]
    out[0]    = [2e 70 fb 8c d5 d8 50 a8 94 38 0e 85 46 9d 33 fc]
    out[1]    = [33 39 da 86 9c a1 f7 1b 3a d0 16 16 ea 42 24 1a]
    out[4]    = [1a 79 cf 13 01 67 2c 52 25 13 8c c8 89 fb 50 72]
    out[16]   = [c8 f2 3f ca 4e 0c 47 46 1a b3 7b 34 1b 57 c7 96]
    out[1023] = [6e 63 21 c1 9b 49 08 57 84 87 14 ea 4f 08 4b 7d]
```