Appeared in "Financial Cryptography — First International Conference FC'97," Anquilla, the British West Indies, 24-28 February 1997, Lecture Notes in Computer Science, Vol.1318, pp.71-89, Springer-Verlag.

The SPEED Cipher

Yuliang Zheng

School of Computing, Monash University
McMahons Road, Frankston, Melbourne, VIC 3199, Australia
Email: yzheng@fcit.monash.edu.au

Abstract. SPEED is a private key block cipher. It supports three variable parameters: (1) data length — the length of a plaintext/ciphertext of SPEED can be 64, 128 or 256 bits. (2) key length — the length of an encryption/decryption key of SPEED can be any integer between 48 and 256 (inclusive) and divisible by 16. (3) rounds — the number of rounds involved in encryption/decryption can be any integer divisible by 4 but not smaller than 32.

SPEED is compact, which is indicated by the fact that the object code of a straightforward implementation of SPEED in the programming language C occupies less than 3 kilo-bytes. It makes full use of current, and more importantly, emerging CPU architectures which host a large number of high-speed hardware registers directly available to application programs. Another important feature of SPEED is that it is built on recent research results on highly nonlinear cryptographic functions, as well as other counter-measures against differential and linear cryptanalytic attacks.

It is hoped that the compactness, high throughput and adjustable parameters offered by SPEED, together with the fact that the cipher is in the public domain, would make it an attractive alternative cipher for security applications including electronic financial transactions.

1 Design Philosophy

The aim of this paper is to introduce a private key cipher that is suitable for software implementation and takes the maximum advantage of emerging computer architectures that host an increasing number of fast internal hardware registers directly available to application programs. The cipher is called SPEED which stands for a Secure Package for Encrypting Electronic Data.

Cryptographic strength of SPEED is built on recent research results on constructing highly nonlinear Boolean functions [15, 16]. Operation efficiency is an important factor that has been taken into account in the process of design. Another design goal is to provide the cipher with applicability to fast one-way hashing and efficient generation of cryptographically strong pseudo-random numbers.

Encryption and pseudo-random number generation have direct applications in providing data confidentiality, whereas one-way hashing is essential for efficient authentication and digital signature.

While most smart cards use 8-bit CPUs, workstations and personal computers are mainly based on 32-bit CPUs which support fast processing of 8, 16 and 32-bit data. Similarly, emerging 64-bit CPUs support efficient handling of 8, 16, 32 and 64-bit data. This results in our decision for the basic data unit for the encryption/decryption operation of SPEED to be a 8-bit, 16-bit or 32-bit word. As a plaintext/ciphertext of SPEED consists of 8 words, choosing a 8-bit word as the basic data unit results in a block cipher on 64-bit data, a 16-bit word results in a block cipher on 128-bit data, and a 32-bit word results in a block cipher on 256-bit data. The process of SPEED is composed of 4 passes, each involving 8 or more consecutive rounds. Thus similarly to RC5 [14], SPEED supports three variable parameters, namely *data length, key length* and *the number of rounds*. Relevant ideas on variable parameters were previously used in a one-way hashing algorithm called HAVAL [18].

A bit-wise nonlinear Boolean operation is employed in each round. To strengthen the cipher against the differential attack proposed by Biham and Shamir [1], a data-dependent cyclic shift is applied on the output of the operation. This technique was inspired by RC5. The use of a maximally nonlinear Boolean function in a bit-wise Boolean operation would help thwart the linear attack discovered by Matsui [9].

The remainder of this paper is organized as follows: Section 2 details the specification of SPEED, Section 3 provides background information on the round transform used in SPEED, and Section 4 discusses the construction and properties of the five nonlinear Boolean functions used in SPEED. A preliminary analysis of the strength of the cipher against cryptanalysis is reported in Section 5, while a comparison of SPEED with other ciphers in terms of its throughput (the number of bits encrypted/decrypted per unit of time) is provided in Section 6. Finally applications of SPEED in one-way hashing and pseudo-random number generation are suggested in Sections 7 and 8.

2 Description of SPEED

First we introduce a few terms used in this paper. As a common practice, a byte is composed of 8 bits. As we mentioned earlier, by a word we mean a string of 8, 16 or 32 bits. All bits in a byte or a word are indexed, starting with 0, from right to left hand side. It is convenient to call right hand side bits *lower bits*, while left hand side bits *upper bits*. Three types of operations are applied to data. The first is bit-wise Boolean operations, the second is cyclic shifts (i.e., rotation) to the right or left, and the third is modular additions.

In the following discussions we use w to indicate the length of (i.e, the number of bits in) a plaintext/ciphertext, ℓ the length of a key, and r the number of rounds. w can be chosen to be 64, 128 or 256, ℓ an integer between 48 and 256 (inclusive) and divisible by 16, and r an integer larger than or equal to 32 and

divisible by 4. SPEED with parameters w, ℓ and r may be denoted by (w, ℓ, r)-SPEED, or simply by w-bit SPEED if the length of a key and the number of rounds are not concerned. In Table 1, various possible combinations of the parameters w, ℓ and r that would provide adequate security are suggested [1]. It is recommended that SPEED with less than 40 rounds be used only for one-way hashing.

plain/ciphertext length w (in bits)	64	128	256
key length ℓ (in bits) ($\ell = 48, 64, ..., 256$, divisible by 16)	≥ 64	≥ 64	≥ 64
number of rounds r ($r = 32, 36, 40, ...,$ divisible by 4)	≥ 64	≥ 48	≥ 48

Table 1. SPEED Parameters for Adequate Security ($r < 48$ may be chosen only when SPEED is used for one-way hashing)

2.1 Encryption

Given a key K of ℓ bits, SPEED scrambles a plaintext M of w bits into a ciphertext C of the same length.

Flow of Data The flow of data in SPEED is depicted in Figure 1. A cryptographic key K, which is a string of ℓ bits, is first expanded by the key scheduling function into four *sub-keys* K_1, K_2, K_3 and K_4. Each K_i, $i = 1, 2, 3, 4$, consists of $\frac{r}{4}$ words or *round keys* where $\frac{r}{4}$ indicates the number of rounds in each pass.

A plaintext M is internally represented as 8 words, each $\frac{w}{8}$ bits. These 8 words are processed by P_1, P_2, P_3 and P_4 consecutively. Each P_i, $i = 1, 2, 3, 4$, is called a pass and involves a sub-key K_i. The output C of P_4 represents the ciphertext of the original plaintext M.

Four Internal Passes As can be seen from Figure 2, the four internal passes P_i, $i = 1, 2, 3, 4$, all operate in a similar fashion, although each pass employs a different sub-key, as well as a different nonlinear function for bit-wise Boolean operations. The four nonlinear bit-wise operations are shown in Table 2 in the form of logic "sum (XOR) of product (AND)".

[1] See also a recent report by Blaze et al [2] which suggests that the length of a key for a private key cipher should be at least 75 to provide adequate security for critical commercial applications.

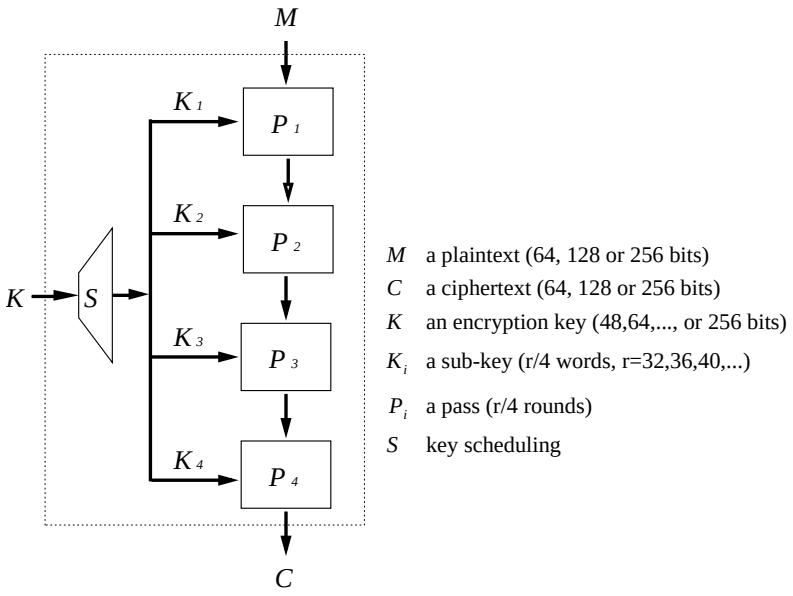


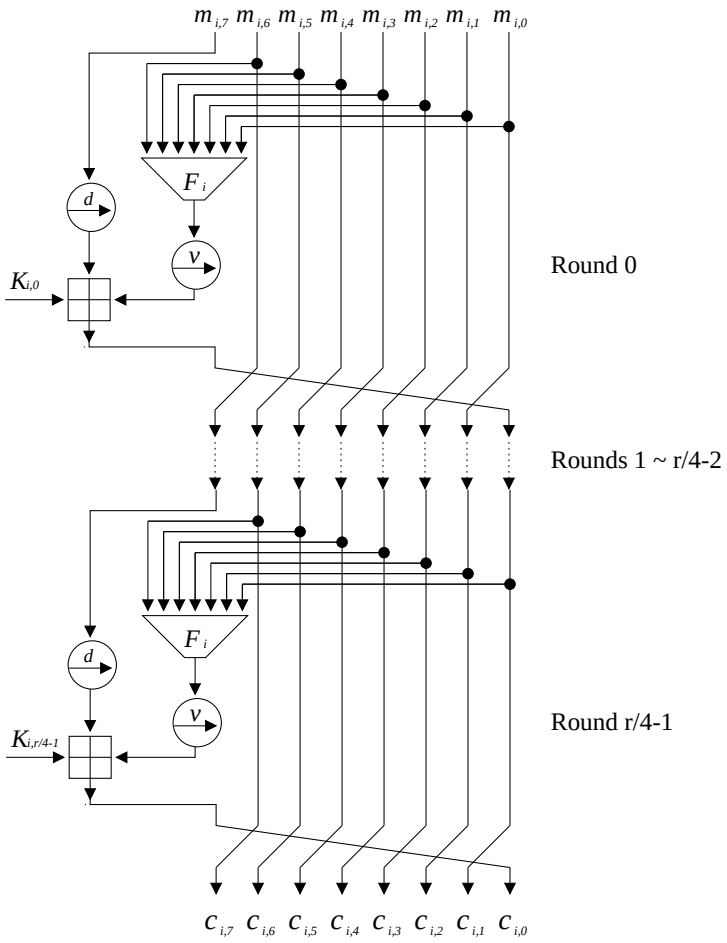
Fig. 1. Encryption Using SPEED

in P_1	$F_1(X_6, X_5, \ldots, X_0) = X_6X_3 \oplus X_5X_1 \oplus X_4X_2 \oplus X_1X_0 \oplus X_0$
in P_2	$F_2(X_6, X_5, \ldots, X_0) = X_6X_4X_0 \oplus X_4X_3X_0 \oplus X_5X_2 \oplus X_4X_3 \oplus X_4X_1 \oplus X_3X_0 \oplus X_1$
in P_3	$F_3(X_6, X_5, \ldots, X_0) = X_5X_4X_0 \oplus X_6X_4 \oplus X_5X_2 \oplus X_3X_0 \oplus X_1X_0 \oplus X_3$
in P_4	$F_4(X_6, X_5, \ldots, X_0) = X_6X_4X_2X_0 \oplus X_6X_5 \oplus X_4X_3 \oplus X_3X_2 \oplus X_1X_0 \oplus X_2$

where
each X_i is a $\frac{w}{8}$-bit word ($w = 64, 128$ or 256),
X_iX_j represents the bit-wise AND, and
$X_i \oplus X_j$ represents the bit-wise XOR of the two words involved.

Table 2. Bit-Wise Nonlinear Boolean Operations Used in P_1, P_2, P_3 and P_4

The input data (a string of 8 words) to P_i is processed in $\frac{r}{4}$ consecutive rounds, each involving the corresponding word in the sub-key K_i, where $i = 1, 2, 3, 4$. In the first round (round 0), the first 7 words in the input are bit-wise processed according to F_i which is shown in Table 2. The result of this operation is then cyclically shifted to right. The exact number of bits by which the result is cyclically shifted is determined by the upper $\log_2 \frac{w}{8}$ bits of the (half-word) sum of the left and right halves of the result. These upper $\log_2 \frac{w}{8}$ bits are indexed by 1, 2 and 3 for $w = 64$, by 4, 5, 6 and 7 for $w = 128$, and by 11, 12, 13, 14 and 15 for $w = 256$. The shifted version of the result of the nonlinear bit-wise operation is added (in the sense of modulo $2^{\frac{w}{8}}$) to the cyclically shifted (to the right by $\frac{w}{16} - 1$ bits) version of the left most word in the input, whose result is



$K_{i,j}$ w/8 bits	d	cyclic shift to the right by d bits, d=w/16-1=3,7 or 15
$m_{i,j}$ w/8 bits	v	cyclic shift to the right by a variable number of bits
$c_{i,j}$ w/8 bits	⊞	addition modulo $2^{w/8}$
w=64, 128 or 256	F_i	bit-wise nonlinear Boolean operation
r = 32,36,40....		

Fig. 2. A Pass P_i in SPEED, $i = 1, 2, 3, 4$

then added to the first word in the sub-key K_i.

The final sum is regarded as an updated version of the left most word in the input to this round. Now the eight words among which the left most one has been updated are rotated to the left by a word, and then used as an input to the next round.

The above process is iterated $\frac{r}{4}$ times, each involving a different word in the sub-key K_i. What follows is pseudo-code for a pass P_i.

A Pass P_i in SPEED, $i = 1, 2, 3, 4$

$t7$, $t6$, ..., $t0$ hold a 8-word data, while $Ki[0]$, ..., $Ki[r/4 - 1]$ hold a $r/4$-word sub-key used in P_i. The contents in $t7$, $t6$, ..., $t0$ are updated according to the following steps:

```
for j from 0 up to (r/4 - 1) do
    t7  = rotate_right(t7, w/16 - 1);
    tmp = Fi(t6, t5, t4, t3, t2, t1, t0);
    vv  = (((tmp >> w/16) + tmp) & HALF_WD_MASK) >> VV_SHIFT;
    tmp = rotate_right(tmp, vv);
    tmp = (t7 + tmp + Ki[j]) & FULL_WD_MASK;

    t7 = t6; t6 = t5; t5 = t4; t4 = t3;
    t3 = t2; t2 = t1; t1 = t0; t0 = tmp;
end of for loop
```

where & denotes bit-wise AND, >> denotes shift-to-right, $rotate\_right(x, n)$ indicates cyclically shifting a $w/8$-bit word x to the right by n bits, FULL_WD_MASK $= 2^{w/8} - 1$, HALF_WD_MASK $= 2^{w/16} - 1$, and VV_SHIFT takes the value of 11 for $w = 256$, 4 for $w = 128$ and 1 for $w = 64$ respectively.

2.2 Key Scheduling

An encryption/decryption key K for SPEED is a binary string of ℓ bits, where ℓ is an integer between 48 and 256 (inclusive) and divisible by 16. The function of the key scheduling is to "extend" K into r words or round keys required by the r rounds of processing. The following issues are considered in designing the key scheduling:

1. It is simple.
2. It allows fast software implementation.
3. It does not have trivial weak keys.
4. It is one-way at least in a weak sense.

The basic data unit in the key scheduling is a double-byte data. Thus an ℓ-bit or $\frac{\ell}{8}$-byte key is first translated into $\frac{\ell}{16}$ internal double-byte data units kb_0,

kb_1, ..., $kb_{\frac{\ell}{16}-1}$. For convenience, let $\ell_{db} = \frac{\ell}{16}$ denote the length of a key in double-bytes. The key scheduling algorithm extends kb_0, kb_1, ..., $kb_{\ell_{db}-1}$. into an array of units kb_0, kb_1, ..., $kb_{\ell_{db}-1}$, $kb_{\ell_{db}}$, ..., kb_{last-1}, where $last$ is $\frac{r}{2}$ when $w = 64$, r when $w = 128$, and $2 * r$ when $w = 256$ respectively. Finally these units are translated into round keys (words) required by the r rounds in SPEED.

Key Scheduling of SPEED

Step 1. Let $kb[0]$, $kb[1]$, ..., $kb[last - 1]$ be an array of double-bytes, where $last$ is $\frac{r}{2}$ when $w = 64$, r when $w = 128$, and $2 * r$ when $w = 256$ respectively. We store the original ℓ-bit key in $kb[0]$, ..., $kb[\ell_{db} - 1]$ as ℓ_{db} double-byte data items. Note that the order of the original key bits is maintained.

Step 2. This step constructs $kb[\ell_{db}]$, ..., $kb[last - 1]$ from the user key data $kb[0]$, ..., $kb[\ell_{db} - 1]$. It employs three double-byte constants $Q_{\ell,0}$, $Q_{\ell,1}$ and $Q_{\ell,2}$.

1. Let $S_0 = Q_{\ell,0}$, $S_1 = Q_{\ell,1}$ and $S_2 = Q_{\ell,2}$.
2. For i from ℓ_{db} to $last - 1$ do the following:
   (a) $T = G(S_2, S_1, S_0)$.
   (b) Rotate T to the right by 5 bits.
   (c) $T = T + S_2 + kb[j] \pmod{2^{16}}$, where $j = i \pmod{\ell_{db}}$.
   (d) $kb[i] = T$.
   (e) $S_2 = S_1$, $S_1 = S_0$, $S_0 = T$.

In the calculation, G represents a bit-wise operation defined by

$$G(S_2, S_1, S_0) = S_2S_1 \oplus S_1S_0 \oplus S_0S_2$$

where each S_i is a double-byte data, S_iS_j represents the bit-wise AND, while $S_i \oplus S_j$ the bit-wise XOR of the two data involved.

Step 3. This step translates the $last$ double-byte data kb_0, kb_1, ..., kb_{last-1} into r rounds keys, each composed of $\frac{w}{8}$ bits. The translation maintains the order of the double-byte data.

The three double-byte constants ($Q_{\ell,0}$, $Q_{\ell,1}$ and $Q_{\ell,2}$) used in the second step are taken from the fractional part of the square root of 15. The first three constants from the fractional part are used for the case of $\ell = 48$, the next three are for $\ell = 64$, and so on. Thus in total 42 constants are required for the 14 different key lengths. These constants are shown below in the hexadecimal form.

```
DF7B D629 E9DB 362F 5D00 F20F C3D1 1FD2 589B 4312 91EB 718E BF2A 1E7D B257 77A6
1654 6B2A 0D9B A9D3 668F 19BE F855 6D98 022D E4E2 D017 EA2F 7572 C3B5 1086 480C
3AA6 9CA0 98F7 D0E4 253C C901 55F3 9BF4 F659 D76C
```

These constants are obtained by using the following Maple program:

```
readlib(write);
open(sqrt15frc);
printlevel := -1;
Digits := 300;
result := evalf(sqrt(15) - 3);
K := 2 ^ 16;
for i from 1 by 1 while i <= 42 do
  nextword := trunc(result * K);
  writeln(convert(nextword,hex));
  result := frac(result * K);
od;
```

Figure 3 illustrates the second step involved in the key scheduling. As can be seen from the figure, the key scheduling algorithm has similarities with an iterative cipher, with a major exception being that it is an irreversible process.

2.3 Decryption

As a private key cipher, SPEED uses the same key both for encryption and decryption. To decrypt a ciphertext C with a key K, the whole process of the algorithm is reversed, except the key scheduling which remains undisturbed. To be more precise, the internal operations of each P_i, $i = 1, 2, 3, 4$, will be conducted in reverse order, which, as depicted in Figure 4, results in $\overline{P}_i$ or the inverse of P_i. The ciphertext C will be processed first by $\overline{P}_4$ with sub-key K_4, followed by $\overline{P}_3$ with K_3, $\overline{P}_2$ with K_2, and finally $\overline{P}_1$ with K_1. The flow of data in decryption is depicted in Figure 5.

3 On the Round Transform Used in SPEED

The widely used Data Encryption Standard or DES [10] was based on a fundamental transform first introduced by Feistel [4, 5]. In its original form, the Feistel transform can be represented by

$$s(L, R) = (R, L \oplus f(R))$$

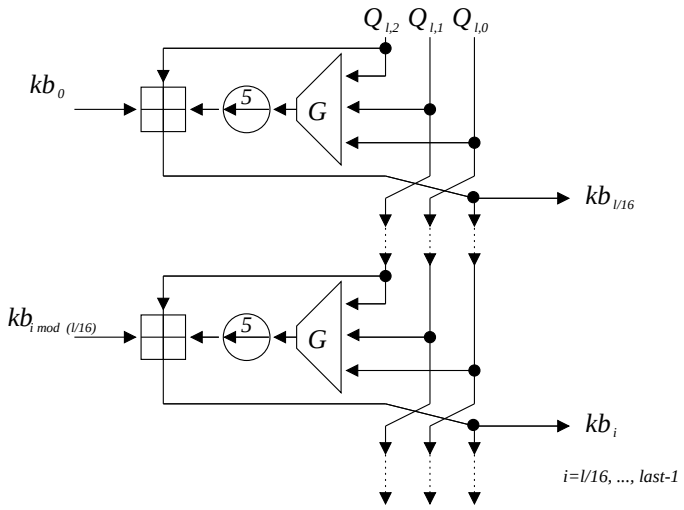
where L and R are binary strings of equal length, $\oplus$ denotes bit-wise XOR and f is a length preserving function.

Using a similar notation, a round in SPEED can be characterized by

$$t(B_7, \ldots, B_2, B_1, B_0) = (B_6, \ldots, B_1, B_0, B_7 \boxplus h(B_6, \ldots, B_1, B_0))$$

where each B_i, $i = 0, 1, \ldots, 7$, is a word of $\frac{w}{8}$ bits, $\boxplus$ denotes addition modulo $2^{\frac{w}{8}}$, and h is a function that shrinks 7 input words into one.

The round transform used in SPEED can be regarded as a generalization of the Feistel transform. In particular, the ideas behind the round transform



5 cyclic shift to the left by 5 bits

addition modulo 2^{16}

G (X2, X1, X0) = X2 X1 ⊕ X1 X0 ⊕ X0 X2

last = 2*r for w=256, r for w=128, and r/2 for w=64

$Q_{l,2}$ $Q_{l,1}$ $Q_{l,0}$ are constants

l is the length of a key

Note: all data are of 16 bits

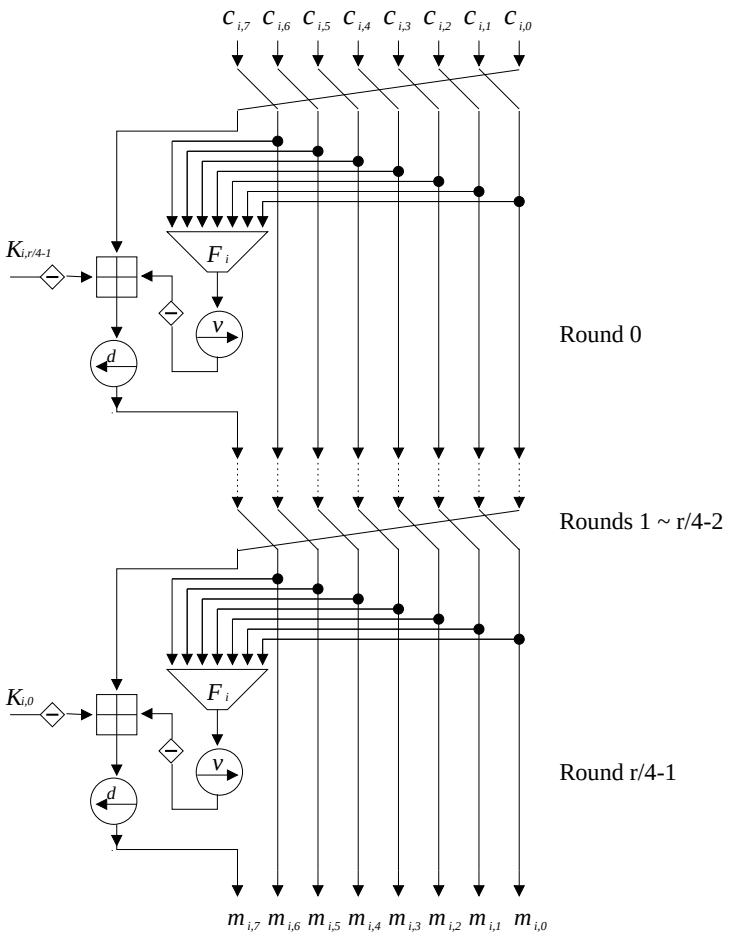
Fig. 3. Step 2 in Key Scheduling

can be traced back to theoretical studies of the Feistel transform carried out by the same author in the late 1980's in [17], where three types of generalized Feistel transforms were suggested, together with a thorough examination of their cryptographic properties. In the terminology of [17], the round transform used in SPEED can be regarded as a "light-weight" version of the inverse of the third type of generalized Feistel transforms.

The round transform in SPEED can be further generalized to

$$t(B_{k-1}, \ldots, B_2, B_1, B_0) = (B_{k-2}, \ldots, B_1, B_0, B_{k-1} \oplus h(B_{k-2}, \ldots, B_1, B_0))$$

for an integer $k \geq 2$. Using a technique similar to that for proving Theorem C1 in [17], one can show that the concatenation of r independent rounds of a transform defined by the transform yields a so-called super-pseudo-random permutation [8] if and only if $r \geq k + 2$. A practical implication of this result is that at least 10 rounds would be required by SPEED, should each round employ a function chosen independently at random.



$K_{i,j}$ w/8 bits

$m_{i,j}$ w/8 bits

$c_{i,j}$ w/8 bits

w = 64, 128 or 256

r = 32,36,40,...

d cyclic shift to the left by d bits, d=w/16-1=3, 7 or 15

v cyclic shift to the right by a variable number of bits

addition modulo $2^{w/8}$

taking the negative (=multiplying by -1)

F_i bit-wise nonlinear Boolean operation

Fig. 4. $\overline{P}_i$, the Inverse of a Pass P_i

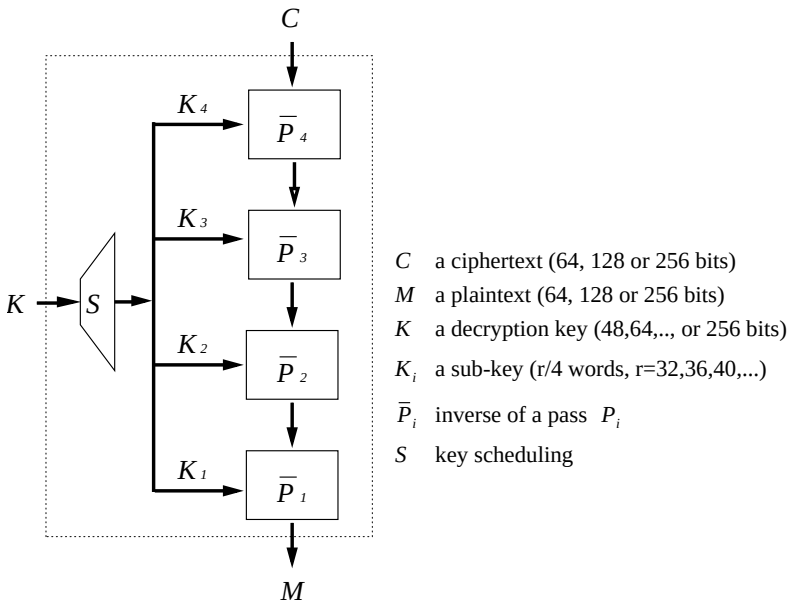


Fig. 5. Decryption Using SPEED

We note that more recently, generalized Feistel transforms have found applications in a number of one-way hashing algorithms, including MD5 [13], SHS [11], HAVAL [18] and other closely related algorithms.

4 Nonlinear Functions Used in SPEED

A function f on V_n, the vector space of dimension n, is said to be nonlinear if it is not affine. The *nonlinearity* of f is defined as the minimum distance from f to all the affine functions. It is known that the nonlinearity of a function on V_n is upper-bounded by $2^{n-1} - 2^{\frac{1}{2}n-1}$. f is said to satisfy the *propagation criterion* with respect to a vector in V_n if complementing an input to f according to the vector results in the output of f to be complemented 50% of the times. Otherwise if it results in the output of f to be a constant (0 or 1), the vector is said to be a *linear structure* of f. In many cryptographic applications, it is desirable for a function to be highly nonlinear, to satisfy the propagation criterion for as many vectors as possible, and to have as few linear structures as possible. As a detailed discussion on nonlinearity is out of the scope of this paper, the reader is referred to [15, 16] for relevant concepts as well as various methods for constructing highly nonlinear functions.

Five nonlinear functions are used in SPEED for bit-wise operations. The first of these functions is used in the key scheduling process, while the other four in the four internal processes P_1, P_2, P_3 and P_4.

4.1 In the Key Scheduling

The nonlinear Boolean function used in the key scheduling can be represented by

$$g(x_2, x_1, x_0) = x_2x_1 \oplus x_1x_0 \oplus x_0x_2$$

g is a balanced majority function with a nonlinearity of 2, which is the maximum value that can be achieved by a function on V_3. It satisfies the propagation criterion with respect to all but one, (1,1,1), non-zero vectors in V_3. The same function was previously used in one-way hashing algorithms SHS [11] and MD4 [12].

4.2 In P_1, P_2, P_3 and P_4

The four nonlinear Boolean functions used in P_1, P_2, P_3 and P_4 are represented by

$$\begin{aligned}
f_1(x_6, x_5, \ldots, x_0) &= x_6x_3 \oplus x_5x_1 \oplus x_4x_2 \oplus x_1x_0 \oplus x_0 \\
f_2(x_6, x_5, \ldots, x_0) &= x_6x_4x_0 \oplus x_4x_3x_0 \oplus x_5x_2 \oplus x_4x_3 \oplus x_4x_1 \oplus x_3x_0 \oplus x_1 \\
f_3(x_6, x_5, \ldots, x_0) &= x_5x_4x_0 \oplus x_6x_4 \oplus x_5x_2 \oplus x_3x_0 \oplus x_1x_0 \oplus x_3 \\
f_4(x_6, x_5, \ldots, x_0) &= x_6x_4x_2x_0 \oplus x_6x_5 \oplus x_4x_3 \oplus x_3x_2 \oplus x_1x_0 \oplus x_2
\end{aligned}$$

Note that f_1, f_3 and f_4, in their original forms, were previously used in a one-way hashing algorithm called HAVAL [18] (in its Passes 1, 3 and 5 respectively). f_2 is constructed using a technique shown in Section 8.3 of [16].

The four functions f_1, f_2, f_3 and f_4 all have very good propagation or avalanche characteristics. In particular,

1. f_1 satisfies the propagation criterion with respect to all but one non-zero vectors in V_7. The vector where the propagation criterion is not satisfied is the only non-zero linear structure of the function.
   The same is true for functions f_3 and f_4.
2. f_2 satisfies the propagation criterion with respect to all but five (5) non-zero vectors in V_7. In contrast to f_1, f_3 and f_4, none of the five vectors where the propagation criterion is not satisfied is a linear structure of f_2. Hence f_2 does not have a non-zero linear structure.

In addition,

1. They achieve the maximum nonlinearity 56 on V_7.
2. They are all balanced.
3. They are inequivalent in the sense that they cannot be converted into one another via a non-singular affine transform on input coordinates. (The inequivalence of f_1, f_3 and f_4 is due to the fact that each has a different algebraic degree. On the other hand, as f_2 has different propagation characteristics, it is equivalent to none of the other three functions.)

4. All non-zero linear combinations of them are balanced. Among the fifteen different combinations, nine achieve the highest nonlinearity 56, three achieve 52, and the other three achieve 48.

It should be added that the coordinates of f_1, f_2, f_3 and f_4 have been re-ordered before they take the current forms. Originally the four functions are as follows:

$$f_1^*(x_6, x_5, \ldots, x_0) = x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus x_0x_1 \oplus x_0$$
$$f_2^*(x_6, x_5, \ldots, x_0) = x_4 \oplus x_4x_6 \oplus x_3x_6 \oplus x_3x_5 \oplus x_2x_5x_6 \oplus x_3x_5x_6 \oplus x_0x_1$$
$$f_3^*(x_6, x_5, \ldots, x_0) = x_1x_2x_3 \oplus x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus x_0x_3 \oplus x_0$$
$$f_4^*(x_6, x_5, \ldots, x_0) = x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus x_0x_1x_2x_3 \oplus x_0x_5 \oplus x_0$$

The coordinates of the four original functions are re-ordered according to Table 3 so that the resulting functions fulfill the requirement that all their non-zero linear combinations are balanced. These re-orderings have been obtained through random sampling.

Function	x_6	x_5	x_4	x_3	x_2	x_1	x_0
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
f_1^*	x_2	x_3	x_5	x_4	x_6	x_1	x_0
f_2^*	x_4	x_0	x_1	x_3	x_6	x_5	x_2
f_3^*	x_1	x_2	x_6	x_0	x_5	x_4	x_3
f_4^*	x_1	x_3	x_5	x_0	x_4	x_6	x_2

Table 3. Re-ordering the Coordinates

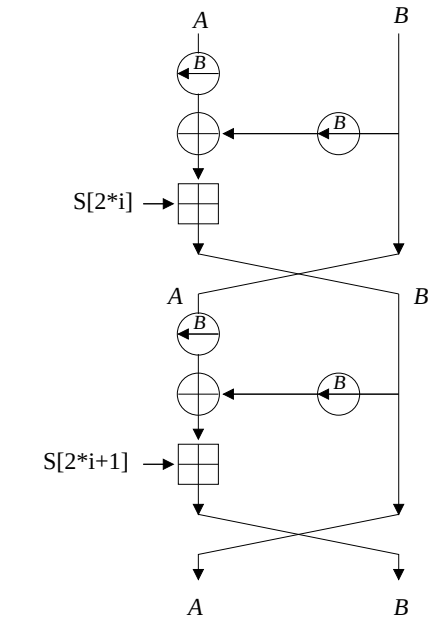
5 Security of SPEED

There are two ciphers that are structurally related to SPEED. These two algorithms are RC5 [14] and MacGuffin [3]. As MacGuffin round transforms are based on a very simple re-arrangement of the substitution boxes (S-boxes) used in DES, SPEED is closer to RC5 than to MacGuffin.

Although SPEED and RC5 share the same feature that both ciphers support three variable parameters, namely *data length, key length* and *the number of rounds*, there are two aspects that differentiate the former from the latter. First, the key scheduling procedures of the two ciphers bear no resemblance. Second, SPEED employs a Boolean function with the maximum nonlinearity in each round, alone with a data-dependent cyclic shift. In contrast, a data-dependent cyclic shift is the only nonlinear operation involved in a round in RC5.

In [6], Kaliski and Yin have presented convincing evidence which suggests that RC5 be secure against both linear and differential attacks if the number of (double) rounds in RC5 is 12 or more. (See also a refined analysis by Knudsen and Meier [7].)

Figure 6 shows a (double) round in RC5 which involves two cyclic shifts and achieves a mixing effect in that both output words are a mixture of both input words. As SPEED has eight words in its input and output data, eight rounds are required to achieve a similar mixing effect, namely each output word is a function of all eight input words. Therefore, structurally a r-round version of SPEED roughly corresponds to a $\frac{r}{8}$-(double) round version of RC5. Due to the use of maximally nonlinear functions in SPEED, we expect that for $r \geq 32$, SPEED is at least as secure as a $\frac{r}{8}$-(double) round RC5. The reader is invited to examine the security of SPEED against all attacks, including linear and differential cryptanalysis.



B cyclic shift to the left by a variable number of bits (determined by the lower bits of B)

S is an array of 2*(r+1) round keys

Fig. 6. A (Double) Round in RC5

Table 1 suggests various possible combinations of the parameters w, ℓ and r that would provide adequate security for commercial applications.

6 Throughput and Compactness of SPEED

It can be seen from Figure 2 that each round involves the following operations:

1. A nonlinear bit-wise operation on 7 words. The exact time for executing it is determined by the complexity of the nonlinear function, and more critically, by the number of fast hardware registers within a CPU which are directly available to a cryptographic application that employs the cipher. The more hardware registers the CPU has, the faster the operation.
2. Two cyclic shift operations. On most machines, they are executed quickly, independently of the number of bits to be cyclically shifted.
3. Two additions modulo $2^{\frac{w}{8}}$. The one which is not shown in Figure 2 is used for finding out the number of bits for the output of the bit-wise operation to be cyclically shifted.

The key scheduling involves additions, bit-wise nonlinear Boolean operations and rotations, each $(last - \frac{\ell}{16})$ times.

A straightforward implementation of SPEED in the programming language C has been carried out. Table 4 shows the throughput (the number of bits encrypted/decrypted per unit of time) of the implementation on a Sun UltraSparc 2 Enterprise (200MHz) as well as on a Pentium Pro 180. Both machines run on the Solaris 2.5.1 operating system. On the UltraSparc the `CC` command calls SparcCompiler C++ 4.1, while on the Pentium it calls ProCompiler C++ 3.01. The throughput indicators have all been obtained for a situation where the key schedule is called only once. The table clearly shows that when the number of rounds are the same, 256-bit SPEED is twice as fast as 128-bit SPEED, and four times as fast as 64-bit SPEED. For comparison, the throughput of the IDEA cipher has also been listed at the bottom of the table.

As the nonlinear Boolean operation in each round of SPEED involves seven words, increase in the throughput of SPEED can be dramatic when it is made parallel by hardware. In addition, pipe-line processing, and more significantly, partial parallel execution of up to six consecutive rounds can be implemented by hardware. This can be seen from the fact that part of the nonlinear Boolean operation in the second round, which is not determined by the outcome of the first round, can be carried out while the first round is being executed. Under a conservative assumption that hardware implementation can be 20 times as fast as its software counterpart on a Sun UltraSparc (200MHz), the throughput of $(256, \ell, 64)$-SPEED, would be boosted to 966 megabit/second. Such a high throughput would be adequate even for applications on future gigabit networks.

Finally, as indicated by the straightforward coding of the cipher in the programming language C, SPEED is suitable for compact implementation either by software or hardware. In particular, when compiled using `'gcc -O2'` on the UltraSparc machine, the object code for the C implementation occupies less than 3 kilo-bytes. Incidentally, this coincides with the size of the object code of a straightforward implementation of the IDEA cipher by R. De Moliner, also in the programming language C.

Instances of SPEED	throughput (megabits/second)			
	on UltraSparc 200		on Pentium Pro 180	
	gcc	CC	gcc	CC
$(256, \ell, 48)$-SPEED	44.91	48.30	27.23	27.83
$(256, \ell, 64)$-SPEED	34.13	36.57	18.96	20.81
$(256, \ell, 80)$-SPEED	27.83	29.77	15.33	16.52
$(256, \ell, 96)$-SPEED	23.06	24.58	12.84	13.69
$(128, \ell, 48)$-SPEED	21.33	24.62	12.43	10.41
$(128, \ell, 64)$-SPEED	16.20	18.55	9.48	7.71
$(128, \ell, 80)$-SPEED	12.93	14.88	7.62	6.15
$(128, \ell, 96)$-SPEED	10.85	12.55	6.37	5.12
$(64, \ell, 64)$-SPEED	8.00	9.28	4.74	5.38
$(64, \ell, 80)$-SPEED	6.46	7.44	3.83	4.32
$(64, \ell, 96)$-SPEED	5.42	6.27	3.22	3.56
IDEA	7.75	16.30	13.64	9.38

(1) Compilers and options:
`'gcc -O2'` on both machines,
`'CC -fast -xO4 -xtarget=ultra'` on UltraSparc, and
`'CC -fast -xO4 -pentium'` on Pentium
(2) IDEA is tested using a package written by R. De Moliner.

Table 4. Throughput of SPEED (and IDEA)

7 Using SPEED in One-Way Hashing

SPEED is a promising candidate for digitally finger-printing or one-way hashing a message of arbitrary length. The length of a finger-print can be up to $w = 256$ bits. It is expected that it is practically infeasible to find two or more different messages that have the same finger-print.

Let $\ell = 256$ and w be an integer larger than or equal to the required number of bits in a finger-print. For the sake of efficiency, r may be chosen from between 32 and 48. For each message M to be hashed, we attach to the end of M three fields. The first field starts with a bit 1 which is followed by zero or more bit 0's so that the length (in bits) of the now-expanded message is 184 modulo 256. The second field has 64 bits indicating the length of M, i.e., the number of bits in M. And finally, the third field consists of 8 bits which indicate the required number of bits in the final finger-print. In what follows it will become clear that since the three fields are attached to the end of M, the operation does not have to be carried out until hashing the last block (of 256 bits or less) in M. This is useful in such a situation as when the length of M is not known beforehand.

Now denote by $M_{n-1}, M_{n-2}, \ldots, M_0$ the padded message, where each M_i consists of 256 bits. The finger-print of the message is obtained in the following

way:

$$D_0 = 0,$$
$$D_{i+1} = D_i + SPEED_{M_i}(D_i), i = 0, 1, \ldots, n-1.$$

The finger-print of the original message M is represented by the desired number of bits in the right hand side of D_n.

Note that in the calculation, $SPEED_{M_i}(D_i)$ should be interpreted as scrambling D_i with M_i as a key, and the summation is word-wise addition modulo $2^{\frac{w}{8}}$.

8 Using SPEED in Pseudo-Random Number Generation

As a block cipher, SPEED can serve as a cryptographically strong pseudo-random number generator when used in the output feedback mode (OFB).

Another simple way to generate cryptographically strong pseudo-random numbers is based on the observation that if SPEED is a strong cipher, then it acts as a pseudo-random function which produces $w = 64$, 128 or 256 pseudo-random bits per application of the algorithm. Let S be a random seed of 256 bits. Assume that IC is a w-bit initial constant value. Then

$$SPEED_S(IC + i), i = 0, 1, 2, \ldots,$$

defines a pseudo-random string that can be used in cryptographic applications. Note that $IC + i$ should be interpreted as $IC + i$ modulo 2^w.

Acknowledgments

It is the author's greatest pleasure to thank the following people: Xian-Mo Zhang for various discussions on nonlinear Boolean functions, Toshiya Itoh and anonymous referees for Financial Cryptography'97 for suggestions that have helped improve the description of the cipher, Kai O'Yang for assistance with smooth landing on Mars (Sparc 10), Jove (Pentium Pro 180) and Saturn (UltraSparc, 200MHz) to test the throughput of SPEED, and finally Lars Knudsen and Hans Dobbertin for helpful comments.

References

1. Biham, E., and Shamir, A. *Differential Cryptanalysis of the Data Encryption Standard*. Springer-Verlag, New York, Heidelberg, Tokyo, 1993.
2. Blaze, M., Diffie, W., Rivest, R., Schneier, B., Shimomura, T., Thompson, E., and Wiener, M. Minimal key length for symmetric ciphers to provide adequate commercial security, January 1996.
3. Blaze, M., and Schneier, B. The MacGuffin block cipher algorithm. In *Fast Software Encryption* (Berlin, New York, Tokyo, 1995), vol. 1008 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 97–110.

4. Feistel, H. Cryptography and computer privacy. *Scientific American 228* (1973), 15–23.
5. Feistel, H., Notz, W. A., and Smith, J. L. Some cryptographic techniques for machine-to-machine data communications. *Proceedings of IEEE 63*, 11 (1975), 1545–1554.
6. Kaliski, B., and Yin, Y. On differential and linear cryptanalysis of the RC5 encryption algorithm. In *Advances in Cryptology - CRYPTO'95* (Berlin, New York, Tokyo, 1995), vol. 963 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 171–184.
7. Knudsen, L., and Meier, W. Improved differential attacks on RC5. In *Advances in Cryptology - CRYPTO'96* (Berlin, New York, Tokyo, 1996), vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 216–228.
8. Luby, M., and Rackoff, C. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing 17*, 2 (1988), 373–386. A preliminary version including other results appeared in the Proceedings of the 18th ACM Symposium on Theory of Computing, 1986, pp.356-363.
9. Matsui, M. Linear cryptanalysis method for DES cipher. In *Advances in Cryptology - EUROCRYPT'93* (1994), vol. 765, Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, New York, pp. 386–397.
10. National Bureau of Standards. Data encryption standard. Federal Information Processing Standards Publication FIPS PUB 46, U.S. Department of Commerce, January 1977.
11. National Institute of Standards and Technology. Secure hash standard. Federal Information Processing Standards Publication FIPS PUB 180-1, U.S. Department of Commerce, April 1995.
12. Rivest, R. The MD4 message digest algorithm, April 1992. Request for Comments (RFC) 1320. (Also presented at Crypto'90, 1990).
13. Rivest, R. The MD5 message digest algorithm, April 1992. Request for Comments (RFC) 1321.
14. Rivest, R. The RC5 encryption algorithm. In *Fast Software Encryption* (Berlin, New York, Tokyo, 1995), vol. 1008 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 86–96.
15. Seberry, J., Zhang, X. M., and Zheng, Y. Nonlinearity and propagation characteristics of balanced boolean functions. *Information and Computation 119*, 1 (1995), 1–13.
16. Zhang, X. M., and Zheng, Y. Characterizing the structures of cryptographic functions satisfying the propagation criterion for almost all vectors. *Design, Codes and Cryptography 7*, 1/2 (1996), 111–134. special issue dedicated to Gus Simmons.
17. Zheng, Y., Matsumoto, T., and Imai, H. On the construction of block ciphers provably secure and not relying on any unproved hypotheses. In *Advances in Cryptology - CRYPTO'89* (Berlin, New York, Tokyo, 1990), vol. 435 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 461–480.
18. Zheng, Y., Pieprzyk, J., and Seberry, J. HAVAL - a one-way hashing algorithm with varialbe length of output. In *Advances in Cryptology - AUSCRYPT'92* (Berlin, New York, Tokyo, 1993), J. Seberry and Y. Zheng, Eds., vol. 718 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 83–104.

Certification Data for SPEED

Data are represented in the hexadecimal form. The byte order used in showing the data is as follows:

the most significant byte $\cdots\cdots$ *the least significant byte*

```
SPEED_DATA_LEN = 64, SPEED_KEY_LEN = 64, SPEED_NO_OF_RND = 64
key        = 00 00 00 00 00 00 00 00
plaintext  = 00 00 00 00 00 00 00 00
ciphertext = 2E 00 80 19 BC 26 85 6D


SPEED_DATA_LEN = 128, SPEED_KEY_LEN = 128, SPEED_NO_OF_RND = 128
key        = FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
plaintext  = FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
ciphertext = 6C 13 E4 B9 C3 17 15 71 AB 54 D8 16 91 5B C4 E8


SPEED_DATA_LEN = 256, SPEED_KEY_LEN = 256, SPEED_NO_OF_RND = 256
key        = 60 5F 5E 5D 5C 5B 5A 59 58 57 56 55 54 53 52 51
             50 4F 4E 4D 4C 4B 4A 49 48 47 46 45 44 43 42 41
plaintext  = 1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10
             0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00
ciphertext = 3D E1 6C FA 9A 62 68 47 43 4E 15 74 69 3F EC 1B
             3F AA 55 8A 29 6B 61 D7 08 B1 31 CC BA 31 10 68
```

This article was processed using the LaTeX macro package with LLNCS style