SURF: SIMPLE UNPREDICTABLE RANDOM FUNCTION

DANIEL J. BERNSTEIN

Abstract. This paper presents surfₙ, a reasonably fast function that converts a 384-bit input into a 256-bit output, given a 1024-bit seed k. When k is secret and uniformly selected, surfₙ seems to be indistinguishable from a uniformly selected 384-bit-to-256-bit function.

1. Introduction

One can use an encryption function as a random function. One can also use a fingerprint as a random function, by seeding the fingerprint appropriately.

These constructions are unsatisfying, for two reasons. First, they are inadequate. Neither a secure encryption function nor a collision-free fingerprint will necessarily produce an unpredictable random function. Encryption functions and fingerprints used in practice do seem to produce unpredictable random functions, but they were not designed to do so.

Second, they are overkill. Unlike an encryption function, a random function can be non-invertible. Unlike a fingerprint, a random function does not need to resist collisions. The constraints of encryption function design and fingerprint design do not apply to random function design.

I attempted in early March 1997 to design an unpredictable random function from scratch. The result is SURF, presented in this paper. For any 1024-bit seed k, the function surfₙ converts any 384-bit input into a 256-bit output; the definition appears in section 2. I do not know any practical algorithm that with nonnegligible probability distinguishes surfₙ from a uniformly selected random function, when k is chosen uniformly. I offer $1000 to the first person to publish a proof that such an algorithm exists, or a proof that no such algorithm exists.

SURF stands for Simple Unpredictable Random Function. SURF takes very little space inside a computer; it can be memorized without trouble. See section 3 for sample code.

See [1, section 2] for a general introduction to unpredictable random functions.

2. Definition

SURF works with 32-bit chunks. It uses three simple operations on these chunks: 
\( x, y \mapsto x + y \), meaning \( x \) plus \( y \) modulo \( 2^{32} \); 
\( x, y \mapsto x \oplus y \), meaning \( x \) exclusive-or \( y \); and 
\( x \mapsto \text{rotate}(x, b) \), meaning \( x \) rotated left by \( b \) bits, with \( b \) constant.

Constants. Define \( a_n = 0 \) for \( 0 \leq n < 12 \), and \( a_n = a_{n-12} + 2654435769 \) for \( n \geq 12 \). Define \( (b_0, b_1, b_2, b_3) = (5, 7, 9, 13) \), and \( b_n = b_{n-4} \) for \( n \geq 4 \).
Construction. Let \( p_0, \ldots, p_{11}, k_0, \ldots, k_{11}, q_0, \ldots, q_{11}, r_0, \ldots, r_7 \) be 32-bit chunks. Define \( k_n = k_{n-12} \) for \( n \geq 12 \). Define \( h_n(x) = ((x^2 \oplus k_n) + a_n) \oplus \text{rotate}(x, b_n) \). Define \( x_n = p_n \oplus q_n \) for \( 0 \leq n < 12 \). Define \( x_n = x_{n-12} + h_n(x_{n-1}) \) for \( 12 \leq n < 396 \). Define \( y_n = r_n \oplus x_{n+196} \oplus x_{n+388} \) for \( 0 \leq n < 8 \). Then \( \text{surf}_k(p_0, \ldots, p_{11}) = (y_0, \ldots, y_7) \) where \( k = (k_0, \ldots, k_{11}, q_0, \ldots, q_{11}, r_0, \ldots, r_7) \).

Example. Say \( k = (0, 1, 2, \ldots, 31) \) and \( p = (0, 0, \ldots, 0) \). Then \((x_0, x_1, \ldots, x_{11}) = (12, 13, \ldots, 23)\); \( x_{12} = 12 + (23 \oplus 0 + a_{12}) \oplus \text{rotate}(23, b_{12}) = 2654436156 \); eventually \( x_{395} = 3769633222 \). The first chunk of \( \text{surf}_k p \) is 1129914649.

Design notes. The security of SURF relies on the use of \( k_0, \ldots, k_{11} \). I added \( q \) and \( r \) simply to annoy the cryptanalyst. (Composing a random function with an independent random invertible function does not reduce security.)

A Feistel sequence is a sequence \( x_0, x_1, \ldots \) satisfying the recurrence \( x_n = x_{n-1} + h_n(x_{n-1}) \), for fixed functions \( h_2, h_3, \ldots \); the map from \((x_0, x_1)\) to \((x_n, x_{n+1})\) is injective. SURF uses a higher-degree recurrence, namely \( x_n = x_{n-12} + h_n(x_{n-1}) \); the map from \((x_0, x_1, \ldots, x_{11})\) to \((x_n, x_{n+1}, \ldots, x_{n+11})\) is injective.

I follow the lead of TEA, introduced by Wheeler and Needham in [5], in doing many rounds of a very fast hash function. SURF, like TEA, hashes 32 times per input chunk; in contrast, DES hashes 8 times per input chunk.

I also follow TEA’s approach to key scheduling, including the choice of constants \( a_n \), namely multiples of 2654435769 = \( 2^{31}(\sqrt{5} - 1) \). See [2, page 510] for some motivation.

I follow standard fingerprint design practice in revealing only a portion of the internal state. (Extracting a portion of the output of a random function does not reduce security.) The sums \( x_{n+196} + x_{n+388} \) should also annoy the cryptanalyst, though perhaps not as much as \( x_{n+192} + x_{n+388} \) would. I originally considered adding two independent sequences, but I decided that for the same investment in time it was better to use a single sequence of twice the length.

I designed the SURF hash function to be as fast as possible subject to two conditions: (1) it alternates between ++ and \( \oplus \); (2) it provides some diffusion of any input change for almost any key. I chose the pattern of rotations \( b_n \) so that a single input bit change would affect every output bit after a few iterations.

SURF’s \( x_n = x_{n-12} + h_n(x_{n-1}) \) structure seems to rule out sparse characteristics: if \( x_0, x_1, \ldots \) and \( x_0', x_1', \ldots \) are two SURF sequences, and if \( x' = x_n \) and \( x'' = x_{n+2} \), then \( x' \) and \( x'' \) are almost forced to be equal too. SURF’s large block size appears to add security here.

I did not bother designing SURF to resist related-seed attacks. Any protocol that ensures integrity will prevent all such attacks.

SURF is immune to timing attacks on typical processors.

3. Implementation

I implemented SURF in portable C and in hand-optimized Pentium assembly language.

The assembly version occupies 636 bytes. It uses 2497 Pentium cycles per call; this means 15.3 million input bits per second, or 10.2 million output bits per second, on a Pentium-100. Setup time is zero.

The C version, compiled with gcc 2.6, occupies 376 bytes and uses 3647 Pentium cycles per call. Again setup time is zero. The code is straightforward:
#define ROT(x,b) (((x) << (b)) | ((x) >> (32 - (b))))
#define MUSH(i,b) x = t[i] += (((x ^ seed[i]) + sum) ^ ROT(x,b));
void surf(out,in,seed)
    uint32 out[8]; uint32 in[12]; uint32 seed[32];
{
    uint32 t[12]; uint32 x; uint32 sum = 0;
    int r; int i; int loop;
    for (i = 0; i < 12; ++i) t[i] = in[i] ^ seed[12 + i];
    for (i = 0; i < 8; ++i) out[i] = seed[24 + i];
    x = t[11];
    for (loop = 0; loop < 2; ++loop) {
        for (r = 0; r < 16; ++r) {
            sum += 0x9e3779b9;
            MUSH(0,5) MUSH(1,7) MUSH(2,9) MUSH(3,13)
            MUSH(4,5) MUSH(5,7) MUSH(6,9) MUSH(7,13)
            MUSH(8,5) MUSH(9,7) MUSH(10,9) MUSH(11,13)
        }
        for (i = 0; i < 8; ++i) out[i] ^= t[i + 4];
    }
}

Here uint32 is a 32-bit unsigned integer type. The entire x sequence is held in the
t array: $x_i, x_{12+i}, x_{24+i}, \ldots$ are stored in $t[i]$.

REFERENCES


DEPARTMENT OF MATHEMATICS, STATISTICS, AND COMPUTER SCIENCE, THE UNIVERSITY OF ILLINOIS AT CHICAGO, CHICAGO, IL 60607-7045
E-mail address: djb@pobox.com