Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 98417, 13 pages
doi:10.1155/2007/98417

Research Article

Pseudorandom Recursions: Small and Fast Pseudorandom Number Generators for Embedded Applications

Laszlo Hars[1] and Gyorgy Petruska[2]

[1] *Seagate Research, 1251 Waterfront Place, Pittsburgh, PA 15222, USA*
[2] *Department of Computer Science, Purdue University Fort Wayne, Fort Wayne, IN 46805, USA*

Received 29 June 2006; Revised 2 November 2006; Accepted 19 November 2006

Recommended by Sandro Bartolini

Many new small and fast pseudorandom number generators are presented, which pass the most common randomness tests. They perform only a few, nonmultiplicative operations for each generated number, use very little memory, therefore, they are ideal for embedded applications. We present general methods to ensure very long cycles and show, how to create super fast, very small ciphers and hash functions from them.

1. INTRODUCTION

For simulations, software tests, communication protocol verifications, Monte-Carlo, and other randomized computations; noise generation, dithering for color reproduction, nonces, keys and initial value generation in cryptography and so forth, many random numbers are needed at high speed. Below we list a large number of pseudorandom number generators. They are so fast and use such short codes that from many applications hardware random number generators can be left out, with all the supporting online tests, whitening and debiasing circuits. If true randomness is needed, a small, slow true random number generator would suffice, which only occasionally provides seeds for the high-speed software generator. This way significant cost savings are possible due to reduced power consumption, circuit size, clock rate, and so forth.

Different applications require different level of randomness, that is, different sets of randomness tests have to pass. For example, at verifying algorithms or generating noise, less randomness is acceptable, for cryptographic applications very complex sequences are needed. Most of the presented pseudorandom number generators take less time for a generated 32-bit unsigned integer than one 32-bit multiplication on most modern computational platforms, where multiplication takes several clock cycles, while addition or logical operations take just one. (There are exceptions, like DSPs and the ARM10 microprocessor. However, their clock speed is constrained by the large and power hungry *single-cycle* multiplication engine.)

Most of the presented pseudorandom number generators pass the Diehard randomness test suite [1]. The ones which fail a few tests can be combined with a very simple function, making all the Diehard tests to pass. If more randomness is needed (higher complexity sequences), a few of these generators can be cascaded, their output sequences can be combined (by addition, exclusive or operation), or one sequence can sample another, and so forth.

Only 32-bit unsigned integer arithmetic is used in this paper (the results of additions or shift operations are always taken modulo 2^{32}). It simplifies the discussions, and the results can easily be converted to signed integers, to long integers, or to floating-point numbers.

There are a large number of fast pseudorandom number generators published, for example, [2–14]. Many of them do not pass the Diehard randomness test suite; others need a lot of computational time and/or memory. Even the well known, very simple, linear congruential generators are slower (see [2]). There are other constructions with good mixing properties, like the RC6 mixer function $x + 2x^2$, [15], or the whole class of invertible mappings, similar to $x + (x^2 \vee 5)$ [16]. They use squaring operations, which make them slower.

In the course of the last year, we coded several thousand pseudorandom number generators and tested them with different seeds and parameters. We discuss here only the best ones found.

2. COMPUTATIONAL PLATFORMS

The presented algorithms use only a few 32-bit arithmetic operations (addition, subtraction, XOR, shift and rotation), which can be performed fast also with 8- or 16-bit microprocessors supporting operations, like add-with-carry. No multiplication or division is used in the algorithms we deal with, because they could take several clock cycles even in 32-bit microprocessors, and/or require large, expensive, and power-hungry hardware cores. We will look at some, more exotic fast instructions, too, like bit or byte reversals. If they are available as processor instructions, they could replace shift or rotation operations.

3. RANDOMNESS TESTS

We used Diehard, the de facto standard randomness test suite [1]. Of course, there are countless other tests one could try, but the large number of tests in the Diehard suite already gives a good indication about the practical usability of the generated sequence. If the randomness requirements are higher, a few of the generators can be combined, with one of the several standard procedures: by cascading, addition, exclusive OR, or one sequence sampling another, and so forth.

The tested properties of the generated sequences do not necessarily change uniformly with the seed (initial value of the generator). In fact, some seeds for some generators are not allowed at all (like 0, when most of the generated sequences are very regular), groups of seeds might provide sequences of similar structure. It would not restrict typical applications of random numbers: sequences resulted from different seeds still consist of very different entries. Therefore, the results of the tests were only checked for pass/fail, we did not test the distribution or independence of the results of the randomness tests over different seeds. Each long sequence in itself, resulted from a given seed, is shown to be indistinguishable from random by a large set of statistical tests, called the Diehard test suite.

Computable sequences, of course, are not truly random. With statistical tests, one can only implicate their suitability to certain sets of applications. Sequences passing the Diehard test suite proved to be adequate for most noncryptographic purposes. Cryptographic applications are treated separately in Sections 8 and 9.

The algorithms and their calling loops were coded in C, compiled and run. In each run, 10 MB of output were written to a binary file, and then the Diehard test suite was executed to analyze the data in the file. The results of the tests were saved in another file, which was opened in an editor, where failed tests (and near fails) were identified.

4. MIXING ITERATIONS

We consider sequences, generated by recursions of the form

$$x_i = f(x_{i-1}, x_{i-2}, \ldots, x_{i-k}). \tag{1}$$

They are called k-stage recursions. We will only use functions of simple structure, built with operations "+," "$\oplus$," "$\ll$," "$\gg$," "$\lll$," and constants. The operands could be in any order, some could occur more than once or not at all, grouped with parentheses. These kinds of iterations are similar to, but more general than the so-called (lagged) Fibonacci recursions. Note the absence of multiplications and divisions.

If the function f is chosen appropriately, the generated sequence will be indistinguishable from true random with commonly used statistical tests. The goals of the constructions are good mixing properties, that is, flipping a bit in the input, all output bits should be affected after a few recursive calls. When we add or XOR shifted variants of an input word, the flipped bit affects a few others in the result. Repeating this with well-chosen shift lengths, all output bits will eventually be affected. If also carry propagation gets into play, the end result is a quite unpredictable mixing of the bits. This is verified with the randomness tests.

4.1. Multiple returned numbers

The random number generator function or the caller program must remember the last k generated numbers (used in the recursion). If we want to avoid the use of (ring) buffers, assigning previously generated numbers to array elements, we could generate k pseudorandom numbers at once. It simplifies the code, but the caller must be able to handle several return values at one call.

The functions are so simple that they can be directly included, inline, in the calling program. If it is desired, a simple wrapper function can be written around the generators, like the following:

```
Rand123(uint32 *a, uint32 *b, uint32 *c)
{
   uint32 x = *a, y = *b, z = *c;
   x += rot(y^z,8);
   y += rot(z^x,8);
   z += rot(x^y,8);
   *a = x; *b = y; *c = z;
}
```

Modern optimizing compilers do not generate code for the instructions of type `x = *a` and `*a = x`, only the data registers are assigned appropriately. If the function is designated as inline, no call-return instructions are generated, either, so optimum speed could still be achieved.

4.2. Cycle length

In most applications it is very important that the generated sequence does not fall into a short cycle. In embedded computing, a cycle length in the order of $2^{32} \approx 4.3 \cdot 10^9$ is often adequate, assuming that different initial values (seeds) yield different sequences. In some applications, many "nonces" are required, which are all different with high probability. If the output domain of the random number generator is n different elements (not necessarily generated in cycle, like when different sequences are combined) and k values are

generated, the probability of a collision (at least two equal numbers) is $0.5\,k^2/n$. (see the appendix). For example, the probability of a collision among a thousand numbers generated by a 32-bit pseudorandom number generator is 0.01%.

4.2.1. Invertible recursion

If, from the recursive equation $x_i = f(x_{i-1}, x_{i-2}, \ldots, x_{i-k})$, we can compute x_{i-k}, knowing the values of $x_i, x_{i-1}, \ldots, x_{i-k+1}$, the generated sequence does not have "ρ" cycles, that is, any long enough generated sequence will eventually return to the initial value, forming an "O" cycle (otherwise there were two inverses of the value, where a short cycle started). In this case, it is easy to determine the cycle lengths empirically: run the iteration in a fast computer and just watch for the initial value to recur. In many applications invertibility is important, for other reasons, too (see [16]).

Most of the multistage generators presented below are easily invertible. One stage recursive generators are more intriguing. Special one stage recursions adding a constant to the XOR of the results of rotations by different amounts are the most common.

$$x_{i+1} = \text{const} + (x_i \lll k_1) \oplus (x_i \lll k_2) \oplus \cdots \oplus (x_i \lll k_m). \tag{2}$$

They are invertible, if we can solve a system of linear equations for the individual bits of the previous recursion value x_i, with the right-hand side formed by the bits of $(x_{i+1} - \text{const})$. Its coefficient matrix is the sum of powers of the unit circulant matrix C: $C^{k_1} + C^{k_2} + \cdots + C^{k_m}$ (here the unit circulant matrix C is a 32×32 matrix containing 0′s except 1′s in the upper-right corner and immediately below the main diagonal, like the 4×4 matrix below).

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{3}$$

If its determinant is odd, there is exactly one solution modulo 2 (XOR is bit-by-bit addition modulo 2). Below we prove that a necessary condition for the invertibility of a one-stage recursion of the above type (2) is that the *number of rotations is odd*.

Lemma 1. *The determinant of M, the sum of k powers of unit circulant matrices is divisible by k.*

Proof. Adding every row of M to the first row (except itself) does not change the determinant. Since every column contains only zeros, except k entries equal to 1 (which may overlap if there are equal powers), all the entries in the first row become k. □

Corollary 1. *Even number of rotations XOR-ed together does not define invertible recursions.*

Proof. The determinant of the corresponding system of linear equations is even, when there is an even number of rotations, according to the lemma. It is 0 modulo 2, therefore the system of equations does not have a unique solution. □

4.2.2. Compound generators

There is no nice theory behind most of the discussed generators, so we do not know the exact length of their cycles, in general. To assure long enough cycles, we take a very different other pseudorandom number generator (which need not be very good), with a known long cycle, and add their output together. The trivial one would be $x_i = i \cdot \text{const} \bmod 2^{32}$ (assuming 32-bit machine words), requiring just one addition per iteration (implemented as `x += const`). It is not a good generator by itself, but for odd constants, like 0x37798849, its cycle is exactly 2^{32} long.

Other very fast pseudorandom number iterations with known long cycles are the Fibonacci generator and the mixed Fibonacci generator (see the appendix). They, too, need only one add or XOR operation for an output, but need two internal registers for storing previous values (or they have to be provided via function parameters). With their least significant bits forming a too regular sequence, they are only suitable as components, when the other generator is of high complexity in those bits.

4.2.3. Counter mode

Another alternative was to choose invertible recursions, and reseed them before each call, with a counter. It guarantees that there is no cycle shorter than the cycle of the counter, which is 2^{160} for a 5-stage generator, far more than any network of computers could ever exhaust. When generating a sequence at 1 GHz rate, even a 64-bit counter will not wrap around for 585 years of continuous operation. There is seldom a practical need for longer cycles than 2^{64}.

Unfortunately, consecutive counter values are very similar, (every odd one differs in just one bit from the previous count) so the mixing properties of the recursion need to be much stronger.

Seeding could be done by the initial counter value, but it is better to use such mixing recursions, which depend on other parameters, too, and seed them with a counter 0, because two sequences with overlapping counter values would be strongly correlated. Furthermore, if this seed is considered a secret key, several of the mixing recursion algorithms discussed below could be modified to provide super fast ciphers. With choosing the complexity of the mixing recursion we could trade speed for security.

4.2.4. Hybrid counter mode

A part of the output of an invertible recursion is replaced with a counter value, and it is used as a new seed for the next call. The feedback values will be very different call by call; thus much fewer recursion steps are enough to achieve sufficient randomness than with pure counter mode. The included counter guarantees different seeds, and so there is no short cycle. It combines the best of two worlds: high speed and guaranteed long cycle.

5. FEEDBACK MODE PSEUDORANDOM RECURSIONS

At Fibonacci type recursions, the most- and least-significant bits of the generated numbers are not very random, so we have to mix in the left-, and right-shifted less regular middle bits to break simple patterns. Some microprocessors perform addition with bit rotation or shift as a combined operation, in one parallel instruction.

It is advantageous to employ both logical and arithmetic operations in the recursion so that the results do not remain in a corresponding finite field (or ring). If they did, the resulting sequences of few-stage generators would usually fail almost all the Diehard tests.

The initial value (seed) of most of these generators must not be all 0, to avoid a fix point.

The algorithms contain several constants. They were found by systematic search procedures, stopped when the desired property (passing all randomness tests in Diehard) was achieved or after a certain number of trials the number of (almost) failed tests did not improve. Below the generators are presented in the order they were discovered. In the conclusions section they are listed in a more systematic order.

5.1. 3-stage generators

If extended precision floating-point numbers (of length $80 \cdots 96$ bit), or single precision triplets (like x, y, z spatial coordinates) are needed, the following generators are very good, giving three 32-bit unsigned integers in each call. For a single-return value, some extra bookkeeping is necessary, like using a ring buffer for the last 3 generated numbers, or moving the newer values to designated variables temp $\leftarrow f(x, y, z)$, $x \leftarrow y$, $y \leftarrow z$, $z \leftarrow$ temp, Return z.

(1) $x_{i+1} = x_{i-2} + (x_{i-1} \ll 8 \oplus x_i \gg 8)$

```
x += y<<8 ^ z>>8;
y += z<<8 ^ x>>8;
z += x<<8 ^ y>>8.
```

This algorithm takes 4 cycles per generated machine word. It can be implemented without any shift operations, just loading the operands from the appropriate byte offset. It is the choice if rotation is not supported in hardware. The recursion is invertible: $x_{i-2} = x_{i+1} - (x_{i-1} \ll 8 \oplus x_i \gg 8)$. Note that using shifts lengths 5 and 3 is slightly more random, but 8 is easier to implement.

(2) Its dual also works (+ and $\oplus$ swapped), with appropriate initial values (not all zeros):

```
x ^= (y<<8) + (z>>8);
y ^= (z<<8) + (x>>8);
z ^= (x<<8) + (y>>8).
```

(3) $x_{i+1} = x_{i-2} + ((x_{i-1} \oplus x_i) \lll 8)$,

```
x += rot(y^z,8);
y += rot(z^x,8);
z += rot(x^y,8).
```

This recursion takes 3 cycles/word. On 8-bit processors, this algorithm, too, can be implemented without any shift operations, just loading the operands from the appropriate byte offset. It is also invertible: $x_{i-2} = x_{i+1} - ((x_{i-1} \oplus x_i) \lll 8)$.

(4) Its dual also works (+ and $\oplus$ swapped), with appropriate initial values:

```
x ^= rot(y+z,8);
y ^= rot(z+x,8);
z ^= rot(x+y,8).
```

(5) $x_{i+1} = x_{i-2} + (x_i \lll 9)$. Its inverse is $x_{i-2} = x_{i+1} - (x_i \lll 9)$:

```
x += rot(z,9);
y += rot(x,9);
z += rot(y,9).
```

This algorithm takes 2 cycles/word, but it cannot be implemented without shift operations.

(6) $x_{i+1} = x_{i-2} + (x_i \lll 24)$ ($\approx$ rotate-right by 8 bits). Its inverse is $x_{i-2} = x_{i+1} - (x_i \lll 24)$:

```
x += rot(z,24);
y += rot(x,24);
z += rot(y,24).
```

It takes also 2 cycles/word. When the processor fetches individual bytes, this algorithm, too, can be implemented without shift operations.

(7) The order of the addition and rotation can be swapped, creating the dual generator:

$x_{i+1} = (x_{i-2} + x_i) \lll 24$ ($\approx$ rotate-right by 8 bits). Its inverse is $x_{i-2} = (x_{i+1} \ll 8) - x_i$:

```
x = rot(x+z,24);
y = rot(y+x,24);
z = rot(z+y,24).
```

This recursion, too, takes 2 cycles/word. With byte fetching, this algorithm can be implemented without shift operations, so, in some sense, these last couple are the best 3-stage generators.

5.2. 4 or more stages

It is straightforward to extend the 3-stage generators to ones of more stages. Here is an example:

(1) $x_{i+1} = (x_{i-3} + x_i) \lll 8$,

```
x = rot(x+w,8);
y = rot(y+x,8);
z = rot(z+y,8);
w = rot(w+z,8).
```

It still uses 2 operations for each generated 32-bit unsigned integer. One could hope that using more stages (larger memory) and appropriate initialization, above a certain size one pseudorandom number could be generated by just one operation. It could be +, −, or $\oplus$. Unfortunately, their low-order bits show very strong regularity. We are not aware of any "small" recursive scheme (with less than a couple dozens stages), which generates a sequence passing all the Diehard tests, and uses only one operation per entry. (Using over 50 stages would make many randomness tests pass, because of the stretched patterns of the low order bits, but the necessary array handling, indexing is more expensive than the computation of the recursion itself.) However, as a component in a

compound generator, a four-stage Fibonacci scheme can be useful. We have to pair it with a recursion, which does not exhibit simple patterns in the low-order bits, that is, which uses shifts or rotations.

(2) On certain (16-bit) processors, swapping the most- and least significant half of a word does not take time (the halves of the operand are loaded in the appropriate order). This would break the regularity of the low order bits, and we can generate a sequence passing the Diehard test suite, with only one addition per entry, in only $k = 5$ stages:

```
for (j = 0; j < k; ++j)
   b[j] += rot(b[(j+2)%5],16).
```

In practice the loop would be unrolled and the rotation operation replaced by the appropriate operand load instruction. We could not find any good 4-stage recursion, which used only shifts or rotations by 16 bits.

5.3. 2-stage generators

In the other direction (using fewer stages), more and more operations are necessary to generate one entry of the pseudorandom sequence, because the internal memory (the number of previous values used in the recursion) is smaller. In general, more computation is necessary to mix the available fewer bits well enough.

The following generator fails only one or two Diehard tests (so it is suitable as a component of a compound generator), with an initial pair of values of $(x, 7)$, with arbitrary seed x.

(1) $x_{i+1} = x_{i-1} + (x_i \ll 8 \oplus x_{i-1} \gg 7)$,

```
x += y<<8 ^ x>>7;
y += x<<8 ^ y>>7.
```

(2) The following variant, using shifts only on byte boundaries, fails a dozen Diehard tests, but as a component generator, it is still usable (all tests passed when combined with a linear sequence):

$x_{i+1} = x_{i-1} + (x_i \ll 8 \oplus x_{i-1} \gg 8)$; $k_{i+1} = k_i +$ 0xAC6D9BB7 mod 2^{32}; $r_i = x_i + k_i$,

```
x += y<<8 ^ x>>8;
y += x<<8 ^ y>>8;
r[0] = x+(k+=0xAC6D9BB7);
r[1] = y+(k+=0xAC6D9BB7);
```

the last two generators are not invertible, so their cycle lengths are harder to determine experimentally. The last generator has a cycle length at least 2^{32} (experiments show much larger values), due to the addition of the linear sequence.

(3) $x_{i+1} = x_{i-1} + (x_i \oplus x_{i-1} \lll 25)$,

```
x += y ^ rot(x,25);
y += x ^ rot(y,25);
```

all tests passed. The complexity of the iteration is 3 cycles/32-bit word. Shift lengths taken only from the set $\{0, 8, 16, 24\}$ do not lead to good pseudorandom sequences (even together with a linear or a Fibonacci sequence), therefore, a true rotate instruction proved to be essential.

(4) If we combine a rotate-by-8 version of this generator, with a mixed two-stage Fibonacci generator, it will pass all the Diehard tests (initialized with x = seed, $y = 1234$ (key); $r = 1, s = 2$):

```
r += s;
s ^= r;

x += y ^ rot(x,8);
y += x ^ rot(y,8);

r[0] = r+x; r[1] = s+y;
```

the *mixed Fibonacci* generator

$$\begin{aligned} x_{2i+1} &= x_{2i-1} + x_{2i}, \\ x_{2i+2} &= x_{2i} \oplus x_{2i+1}, \end{aligned} \tag{4}$$

with initial values $\{1, 2\}$ has a period of $3 \cdot 2^{30} \approx 3.2 \cdot 10^9$ (see the appendix). It is easily invertible, and $6.5 \cdot 10^9$ values are generated before they start to repeat. The low-order bits are very regular, but it is still suitable as a component in a compound generator, as above.

5.4. 1-stage generators

We have to apply some measures to avoid fix points or short cycles at certain seeds. An additive constant works. Alternatively, one could continuously check if a short cycle occurs, but this check consumes more execution time than adding a constant, which prevents short cycles.

(1) $x_{i+1} = x_i \oplus (x_i \lll 5) \oplus (x_i \lll 24) +$ 0x37798849,

```
x = (x ^ rot(x,5) ^ rot(x,24)) + 0x37798849.
```

This generator takes 5 cycles/32-bit word, still less than half of a single multiplication time on the Pentium microprocessor. Unfortunately, shift lengths taken from the set $\{0, 8, 16, 24\}$ do not lead to good pseudorandom sequences, therefore, for efficient implementation of this generator the processor must be able to perform fast shift instructions. If we add the linear sequence $k_{i+1} = k_i +$ 0xAC6D9BB7 mod 2^{32} to the result $r_i = x_i + k_i$, it improves the randomness and makes sure that the period is at least 2^{32}. The pure recursive version is invertible, because the determinant of the system of equations on the individual bits is odd (65535).

The last recursion can be written with shifts instead of rotations:

```
x = (x ^ x<<5 ^ x>>27 ^ x<<24 ^ x>>8)
    + 0x37798849.
```

It takes 9 cycles/32-bit result, still faster than one multiplication.

(2) On certain microprocessors, shifts with 24 or 8 bits can be implemented with just appropriately addressing the data, so shifts on byte boundaries are advantageous:

```
x = (x ^ x<<8 ^ x>>27 ^ x<<24 ^ x>>8)
    + 0x37798849.
```

It works, too, (passing all the Diehard tests) with one more shift on byte boundaries, but the corresponding determinant is even (256), so the recursion is not invertible.

```
(3)  x = (x ^ x<<5 ^ x>>4 ^ x<<10 ^ x>>16)
         + 0x41010101.
```

With this generator, only one Diehard test fails. It takes 9 cycles/32-bit word. On 16-bit microprocessors, some work can be saved, because $x \gg 16$ merely accesses the most significant word of the operand. It is faster than one (Pentium) multiplication and invertible, with odd determinant = 114717.

(4) With little loss of randomness, we can drop a shifted term:

```
x = (x ^ x<<5 ^ x<<23 ^ x>>8) + 0x55555555.
```

Seven Diehard tests fail, but it is still suitable as a component generator (even with the linear sequence $x_i = i \cdot$ 0x37798849 mod 2^{32}). It takes 7 cycles/32-bit word. One cycle can be saved at 8-bit processors, because $x \gg 8$ just accesses the three most significant bytes of the operand. It is invertible with odd determinant = 18271.

(5) If we want one more shift operation to be on byte boundaries, we can use

```
x = (x ^ x<<5 ^ x<<24 ^ x>>8) + 0x6969F969.
```

Here nine Diehard tests fail, but it is still suitable as a component RNG (even with the very simple $x_i = i \cdot$ 0xAC5532BB mod 2^{32}). It is *not* invertible, having an even determinant = 16038.

5.5. Special CPU instructions

There are many other less frequently used microprocessor instructions, like counting the 1-bits in a machine word (Hamming weight), finding the number of trailing or leading 0-bits (Intel Pentium: BSFL, BSRL instructions). They would allow variable shift lengths in recursions, but in a random looking sequence the number of leading or trailing 0 or 1 bits are small, so there is no much variability in them. Also, it is easy to make a mistake, like adding its Hamming weight to the result, what actually makes the sequence less random.

Some microprocessors offer a bit-reversal instruction (used with fast Fourier transforms) or byte-reversal (Intel Pentium: BSWAP), to handle big- and little-endian-coded numeric data. They can be utilized for pseudorandom number generation, although they do not seem to be better than rotations. These instructions are most useful, if they do not take extra time (like only the addressing mode of the operands needs to be appropriately specified, or the addressing mode can be set separately for a block of data).

(1) An example is the following feedback mode pseudorandom number generator:

```
x = RevBytes(x+z);
y = RevBytes(y+w);
z = RevBytes(z+r);
w = RevBytes(w+x);
r = RevBytes(r+y);
```

this 5-stage-lagged Fibonacci type generator is invertible, passes all the Diehard tests, and needs only one addition per iteration. The operands are stored in memory in one (little- or big endian) coding, and loaded in different byte order. This normally does not take an extra instruction, so this generator is the possible fastest for these platforms. (Note that no such 4-stage generators are found, which pass all the Diehard tests, and perform one operation per iteration together with byte or bit reversals. Not even when bit and byte reversals are intermixed.)

6. COUNTER MODE: MIXER RECURSIONS AND PSEUDORANDOM PERMUTATIONS

Invertible recursions, reinitialized with a counter at each call, yield a cycle as long as the period of the counter. For practical embedded applications, 32-bit counters often provide long enough periods, but we also present pseudorandom recursions with 64-bit and 128-bit counters. The corresponding cycle lengths are sufficient even for very demanding applications (like huge simulations used for weather forecast or random search for cryptographic keys).

If the counter is not started from 0 but from a large seed, these generators provide different sequences, without simple correlations. Also, in some applications it is necessary to access the pseudorandom numbers out of order, which is very easy in counter mode, while hard with other modes.

6.1. 1-stage generators

(1) With the parameters (L,R,A) = (5, 3, 0x95955959), the following recursion provides a pseudorandom sequence, which passes all Diehard tests, without near fails (p = 0.999+):

```
x = k++;
x = (x ^ x<<L ^ x>>R) + A;
x = (x ^ x<<L ^ x>>R) + A;
x = (x ^ x<<L ^ x>>R) + A;
x = (x ^ x<<L ^ x>>R) + A;
x = (x ^ x<<L ^ x>>R) + A;
x = (x ^ x<<L ^ x>>R) + A;
x = (x ^ x<<L ^ x>>R);
```

(2) if shifts only on byte boundaries are used, we need 12 iterations (instead of the 7 above), the last one without adding A. The parameters are (L,R,A) = (8, 8, 0x9E3779B9). There is no p = 0.999+ in the Diehard tests, which gives some assurances that any initial counter value works.

(3) With rotations, the parameters (L,R,A) = (5, 9, 0x49A8D5B3) give a faster generator, with only one p = 0.999+ in Diehard:

```
x = k++;
x = (x ^ rot(x,L) ^ rot(x,R)) + A;
x = (x ^ rot(x,L) ^ rot(x,R)) + A;
x = (x ^ rot(x,L) ^ rot(x,R)) + A;
x = (x ^ rot(x,L) ^ rot(x,R));
x = (x ^ rot(x,L) ^ rot(x,R));
```

(4) if rotations only on byte boundaries are used, we need 9 iterations (instead of the 5 above), the last two without adding A: (L,R,A) = (8, 16, 0x49A8D5B3) two p = 0.999+ in Diehard.

6.2. 2-stage generators

In this case, the longer counter (64-bit) makes the input more correlated, and so more computation is needed to mix the bits well enough, but we get two words at a time. Different parameter sets lead to different pseudorandom sequences, similar in randomness and speed (9 iterations):

(1) (L,R,A,B,C) = (5, 3, 0x22721DEA, 6, 3) no p = 0.999+ in Diehard.
(2) (L,R,A,B,C) = (5, 4, 0xDC00C2BB, 6, 3) one p = 0.999+ in Diehard.
(3) (L,R,A,B,C) = (5, 6, 0xDC00C2BB, 6, 3) no p = 0.999+ in Diehard.
(4) (L,R,A,B,C) = (5, 7, 0x95955959, 6, 3) no p = 0.999+ in Diehard.

```
x = k++; y = 0;
for (j = 0; j < B; j+=2) {
   x += (y ^ y<<L ^ y>>R) + A;
   y += (x ^ x<<L ^ x>>R) + A;
}
for (j = 0;;) {
   if (++j > C) break;
   x += y ^ y<<L ^ y>>R;
   if (++j > C) break;
   y += x ^ x<<L ^ x>>R;
}
```

If shifts only on byte boundaries are used, we needed only slightly more, 11 iterations, the last three without adding A.

(5) (L,R,A,B,C) = (8, 8, 0xDC00C2BB, 8, 3) one p = 0.999+ in Diehard.
Again, with rotations fewer iterations are enough. The following recursions generate different pseudorandom sequences, similar in randomness and in speed (7 iterations).
(6) (L,R,A,B,C) = (5,24, 0x9E3779B9, 4, 3) no 0.999+ in Diehard.
(7) (L,R,A,B,C) = (7,11, 0x9E3779B9, 4, 3) no 0.999+ in Diehard.
(8) (L,R,A,B,C) = (5,11, 0x9E3779B9, 4, 3) no 0.999+ in Diehard.
(9) (L,R,A,B,C) = (5, 9, 0x49A8D5B3, 4, 3) no 0.999+ in Diehard.
(10) (L,R,A,B,C) = (5, 8, 0x22721DEA, 4, 3) no 0.999+ in Diehard.

```
x = k++; y = 0;
for (j = 0; j < B; j+=2) {
   x += (y ^ rot(y,L) ^ rot(y,R)) + A;
   y += (x ^ rot(x,L) ^ rot(x,R)) + A;
}
for (j = 0;;) {
   if (++j > C) break;
   x += y ^ rot(y,L) ^ rot(y,R);
   if (++j > C) break;
   y += x ^ rot(x,L) ^ rot(x,R);
}
```

If rotations only on byte boundaries are used, we needed 10 iterations (instead of the 7 above), the last two without adding A.

(11) (L,R,A,B,C) = (8, 16, 0x55D19BF7, 8, 2) two 0.999+ in Diehard.

Recursions with rotation by 8 and 24 need one more iteration.

6.3. 4-stage generators

These generators mix even longer counters (128 bit) containing correlated values, so still more computation is needed to mix the bits well enough, but 4 pseudorandom words are generated at a time. Different parameter sets lead to different pseudorandom sequences, similar in randomness and in speed (11 iterations):

```
x = k++; y = 0; z = 0; w = 0;
for (j = 0; j < B; j+=4) {
   x += ((y^z^w)<<L) + ((y^z^w)>>R) + A;
   y += ((z^w^x)<<L) + ((z^w^x)>>R) + A;
   z += ((w^x^y)<<L) + ((w^x^y)>>R) + A;
   w += ((x^y^z)<<L) + ((x^y^z)>>R) + A;
}
for (j = 0;;) {
   if (++j > C) break;
   x += ((y^z^w)<<L) + ((y^z^w)>>R);
   if (++j > C) break;
   y += ((z^w^x)<<L) + ((z^w^x)>>R);
   if (++j > C) break;
   z += ((w^x^y)<<L) + ((w^x^y)>>R);
   if (++j > C) break;
   w += ((x^y^z)<<L) + ((x^y^z)>>R);
}
```

(This code is for experimenting only. In real-life implementations loops are unrolled.)

(1) (L,R,A,B,C) = (5, 3, 0x95A55AE9, 8, 3) no 0.999+ in Diehard.

(2) (L,R,A,B,C) = (5, 4, 0x49A8D5B3, 8, 3) no 0.999+ in Diehard, and several similar ones.

(3) (L,R,A,B,C) = (5, 7, 0xDC00C2BB,8, 3) no 0.999+ in Diehard.

Common expressions could be saved and reused, done automatically by optimizing compilers. If shifts only on byte boundaries are used, we needed only slightly more, 13 steps (instead of the 11 above), the last one without adding A.

(4) (L,R,A,B,C) = (8, 8, 0x49A8D5B3, 12, 1) no 0.999+ in Diehard.

Here, also, rotations allow using simpler recursive expressions. The following ones generate different pseudorandom sequences, similar in randomness and in speed (13 steps):

(5) (L,R,A,B,C) = (5, -, 0x22721DEA, 12, 1) no 0.999+ in Diehard.

(6) (L,R,A,B,C) = (9, -, 0x49A8D5B3, 12, 1) no 0.999+ in Diehard.

```
x = k++; y = 0; z = 0; w = 0;
for (j = 0; j < B; j+=4) {
   x += rot(y^z^w,L) + A;
   y += rot(z^w^x,L) + A;
   z += rot(w^x^y,L) + A;
   w += rot(x^y^z,L) + A;
}
for (j = 0;;) {
   if (++j > C) break;
   x += rot(y^z^w,L);
   if (++j > C) break;
   y += rot(z^w^x,L);
   if (++j > C) break;
   z += rot(w^x^y,L);
   if (++j > C) break;
   w += rot(x^y^z,L);
}
```

(This code is for experimenting only. In real-life implementations loops are unrolled.) If rotations only on byte boundaries are used, we needed 15 steps (instead of the 13 above), the last three without adding A.

(7) (L,R,A,B,C) = (8, -, 0x95A55AE9, 12, 3) no 0.999+ in Diehard.

The dual recursion (swap "+" and "⊕") is very similar in both running time and randomness:

```
x = k++; y = 0; z = 0; w = 0;
for (j = 0; j < B; j+=4) {
   x ^= rot(y+z+w,L) ^ A;
   y ^= rot(z+w+x,L) ^ A;
   z ^= rot(w+x+y,L) ^ A;
   w ^= rot(x+y+z,L) ^ A;
}
for (j = 0;;) {
   if (++j > C) break;
   x ^= rot(y+z+w,L);
   if (++j > C) break;
   y ^= rot(z+w+x,L);
   if (++j > C) break;
   z ^= rot(w+x+y,L);
   if (++j > C) break;
   w ^= rot(x+y+z,L);
}
```

(8) (L,R,A,B,C) = (5, -, 0x95955959, 12, 1) no 0.999+ in Diehard.

(9) (L,R,A,B,C) = (6, -, 0x95955959, 12, 1) no 0.999+ in Diehard.

(10) (L,R,A,B,C) = (7, -, 0x95955959, 12, 1) no 0.999+ in Diehard.

(11) (L,R,A,B,C) = (9, -, 0x95955959, 12, 1) no 0.999+ in Diehard.

If rotations only on byte boundaries are used, similar to the dual recursions, we needed 15 steps (instead of the 13 above), the last three without adding A.

(12) (L,R,A,B,C) = (8, -, 0x95955959, 12, 3) no 0.999+ in Diehard.

Other combinations of "+" and "⊕" are also similar, leading to different families of similar generators:

```
     x += rot(y+z+w,L) ^ A;...
or   x ^= rot(y+z+w,L) + A;...
```

However, when only "+" or only "⊕" operations are used, the resulting sequences are poor.

7. HYBRID COUNTER MODE

If we split a machine word the recursion operates on, for the counter and for the output feedback value, the guaranteed cycle length of the resulting sequence will be too short. Therefore, one stage is not enough.

7.1. 2-stage generators

```
    x = k++;
(1) x += ((x^y)<<11) + ((x^y)>>5) ^ y;
    y += ((x^y)<<11) + ((x^y)>>5) ^ x;
```

it needs 6 cycles/word. All Diehard tests are passed, with only one 0.999+. Other combinations of + and ⊕ give similar results, as long as both operations are used.

A slightly slower (8 cycles) and slightly better (no near fail) 2-stage generator is the following:

```
    x = k++;
(2) x += x<<5 ^ x>>7 ^ y<<10 ^ y>>5;
    y += y<<5 ^ y>>7 ^ x<<10 ^ x>>5;
```

shift on byte boundaries with 8 cycles/word:

```
    x = k++;
(3) x += (y<<8) ^ ((x^y)<<16) ^ ((x^y)>>8)+y;
    y += (x<<8) ^ ((x^y)<<16) ^ ((x^y)>>8)+x;
```

with rotations only half as much work is needed (4 cycles/word):

```
    x = k++;
(4) x += rot(x,16) ^ rot(y,5);
    y += rot(y,16) ^ rot(x,5);
```

its dual is equally good (no near fails in Diehard), but requires a slightly different rotation length.

```
    x = k++;
(5) x ^= rot(x,16) + rot(y,7);
    y ^= rot(y,16) + rot(x,7);
```

the following recursion is the same for x, and for y, and uses rotations only on byte boundaries. It uses 6 operations/word (common subexpressions reused), 2 more than the recursions above.

```
    x = k++;
(6) x ^= rot(x+y,16) + rot(y+x,8) + y+x;
    y ^= rot(y+x,16) + rot(x+y,8) + x+y;
```

swapping some + and ⊕ operations the resulting recursion is equally good (no Diehard test fails, no p = 0.999+):

```
    x = k++;
(7) x += (rot(x^y,16) ^ rot(y^x,8)) + (y^x);
    y += (rot(y^x,16) ^ rot(x^y,8)) + (x^y);
```

7.2. 3-stage generators

These generators are at most 1 instruction longer than the corresponding pure feedback mode generators, but still there is not even a near fail in the Diehard tests:

```
    x = k++;
(1) x += z ^ y<<8 ^ z>>8;
    y += x ^ z<<8 ^ x>>8;
    z += y ^ x<<8 ^ y>>8;
```

Its dual is equally good:

```
    x = k++;
(2) x ^= z + (y<<8) + (z>>8);
    y ^= x + (z<<8) + (x>>8);
    z ^= y + (x<<8) + (y>>8);
```

The following feedback mode generator with *rotations* works unchanged in hybrid counter mode:

```
    x = k++;
(3) x += rot(y^z,8);
    y += rot(z^x,8);
    z += rot(x^y,8);
```

like its dual

```
    x = k++;
(4) x ^= rot(y+z,8);
    y ^= rot(z+x,8);
    z ^= rot(x+y,8);
```

The generator below is faster (2 cycles/word), but uses an odd-length rotation and has one near fail in the Diehard tests:

```
    x = k++;
(5) x += rot(y,9);
    y += rot(z,9);
    z += rot(x,9);
```

7.3. 4-stage generator

A variant of the simplest feedback mode generator works in hybrid counter mode, too, without near fails in Diehard (no p = 0.999+). The rotations are on byte boundaries.

```
    x = k++;
    x = rot(x+y,8);
(1) y = rot(y+z,8);
    z = rot(z+w,8);
    w = rot(w+x,8);
```

7.4. 6-stage generator with byte reversal

With only one arithmetic instruction per iteration, 5 stages are not enough to satisfy all the Diehard tests, but a variant of the feedback mode 6-stage generator works in the hybrid counter mode, too, without near fails in Diehard (p = 0.999+):

```
    x = k++;
    x = RevBytes(x+y);
    y = RevBytes(y+z);
(1) z = RevBytes(z+w);
    w = RevBytes(w+r);
    r = RevBytes(r+s);
    s = RevBytes(s+x);
```

8. CIPHERS

Counter-mode pseudorandom recursions can be used as very simple, super fast ciphers, when the security requirements are not high, like at RFID tags tracking merchandise in a warehouse.

8.1. Four-way feistel network

We need to use many more rounds than the minimum listed above, because they only guarantee a certain set of randomness tests (Diehard) to pass. Instead of adding a constant in each round, we add a number derived from the encryption key by another pseudorandom recursion. These form a small set of subkeys, called key schedule. They are computed in an initialization phase, about the same complexity as the encryption of one data block. At decryption, the same key schedule is needed, and the inverse recursion is computed backwards.

If the subkey used in a particular round is fixed, a certain type of attack is possible: match round data from different rounds [17]. To prevent that, the subkeys are chosen data dependently. It provides more variability than only assuring that each round is different, which was a design decision, among others, in the TEA cipher, and its improvements [18–20]. However, many different subkeys require larger memory, and could necessitate swapping subkeys in and out of the processor cache, which poses security risks. To combat this problem, one can recompute the subkeys on the fly, maybe, with some precomputed data to speed up this subkey generation. Here is an example key schedule, continuing the initial key sequence $k_0, \ldots, k_3$:

```
for(j = 4; j<16; ++j)
   k[j] = k[j-4] ^ rot(k[j-3]+k[j-2]+k[j-1],5)
                 ^ 0x95A55AE9;
```

Block lengths can be chosen as any multiple of 32 bits, as described in the Block-TEA and the XXTEA algorithms [20]. We present an example with 128-bit blocks $\{x, y, z, w\}$ and 128-bit keys $k_0, \ldots, k_3$. 16 subkeys are computed in advance. (They are reused for encrypting other data.) One can use the original keys only (2-bit index), or generate many subkeys, as desired. The more subkeys, the less predictable the behavior

of the encryption algorithm, but also the more memory used. Subkey selection can be performed by the least significant, or any other data bits like `k[x&15]` or `k[x≫28]`, and so forth. Consecutive subkeys are strongly correlated, but the order in which they are used is unpredictable. With more work, one can make the subkeys less correlated: perform a few more iterations before they get stored, or the subkeys could be generated as sums of different pseudorandom sequences. Here is a very simple cipher according to the design above:

```
for (j = 0; j < 8; ++j) {
   x += rot(y^z^w,9) + k[y>>28];
   y += rot(z^w^x,9) + k[z>>28];
   z += rot(w^x^y,9) + k[w>>28];
   w += rot(x^y^z,9) + k[x>>28]; }
```

A similar function wrapper could be used around the instructions, as described in the iterations section. The number of rounds has to be large enough that a single input bitflip has an affect on any output bit, and so *differential cryptanalysis would fail*. A bitflip in w changes a bit in x, and after the rotation y has already at least 2 affected bits. Similarly, z has at least 3 bits changed in the first round, and when w is updated at least 6 of its bits are affected. In the second round it gets to 36, more than the 32, present in a machine word, therefore, 2 rounds already mix the bits of w sufficiently. For the same effect on x one more round is needed, so 3 rounds perform a good enough mixing. It is consistent to the results in the counter mode section above. For higher security (less chance for some exploitable regularity) one should go with more rounds, probably 16 or even 32. The example above uses 8 rounds, which is very fast but somewhat risky.

Decryption goes backward in the recursion, the natural way, after generating the same subkeys:

```
for (j = 8; j > 0; --j) {
   w -= rot(x^y^z,9) + k[x>>28];
   z -= rot(w^x^y,9) + k[w>>28];
   y -= rot(z^w^x,9) + k[z>>28];
   x -= rot(y^z^w,9) + k[y>>28]; }
```

8.2. Even-Mansour construction

In [21] a block cipher construction was presented, which makes use of a publicly known permutation F, where it is easy to compute $F(X)$ and $F^{-1}(X)$ for any given input $X \in \{0;1\}^n$. The key consists of two n-bit subkeys K_1 and K_2. The ciphertext C of the plaintext P is defined by

$$C = K_2 \oplus F(P \oplus K_1). \tag{5}$$

Decryption is done by solving the above equation for P:

$$P = K_1 \oplus F^{-1}(C \oplus K_2). \tag{6}$$

This scheme is secure if F is a good mixing function ($\sim$ pseudorandom permutation). Here we can use a function defined by any of our counter mode pseudorandom recursions.

9. MIXING AND HASH FUNCTIONS

In a counter mode pseudorandom recursion, the counter value could be replaced by arbitrary input. The result is a good mix of the input bits. In the case of hash functions, we do not want invertibility. The easiest to achieve noninvertibility is to compute mix values of two or more different blocks of data, and add them together. This provides a compression function. Hash functions can be built from them by well-known constructions, Merkle-Damgård (see [22, 23]), Davies-Meyer, Double-Pipe hash construction (see [24, 25]) and their combinations. See also [26].

10. CONCLUSIONS

We presented many small and fast pseudorandom number generators, which are suitable to most embedded applications, directly. For cryptography (ciphers, hash function), they have to be applied via known secure constructions, like the ones described in Sections 8 and 9. We list all the generators in Tables 1, 2, and 3 on their modes of operation, sorted by the size of the used memory. The algorithms are referenced by their number in the corresponding subsection (for the appropriate number of stages).

A.1. Collision probability

Choose k elements randomly (repetition allowed) from n different ones. The probability of no collision (each element is chosen only once), for any (n,k) pair:

$$P(n,k) = \frac{n(n-1)\cdots(n-k+1)}{n^k} = \frac{n!}{(n-k)!\,n^k} \approx \frac{(n/e)^n\sqrt{2\pi n}}{((n-k)/e)^{n-k}\sqrt{2\pi(n-k)}\,n^k} = \left(\frac{n}{n-k}\right)^{n-k+1/2} e^{-k}. \tag{A.1}$$

(Stirling's approximation is applied to the factorials.) To avoid computing huge powers, take the logarithm of the last expression. The exponential of the result is $P \approx e^{(n-k+(1/2))\cdot\log(n/(n-k))-k}$. A 2-term Taylor expansion $\log(1+x) \approx x - x^2/2$ with the small $x = k/(n-k)$, yields $-k((2n(k-1)-2k^2+3k)/4(n-k)^2)$ in the exponent. Keeping only the dominant terms (assuming $n \gg k \gg 1$) we get the approximation $P \approx e^{-k^2/2n}$, for the probability that all items are different. If the exponent is small ($k^2 \ll n$), with applying a two-term Taylor expansion $e^x \approx 1+x$ the probability of a *collision* is well approximated by

$$1 - P \approx \frac{k^2}{2n}. \tag{A.2}$$

A.2. Mixed Fibonacci generator

$$\begin{aligned} x_{2i+1} &= x_{2i-1} + x_{2i}, \\ x_{2i+2} &= x_{2i} \oplus x_{2i+1}. \end{aligned} \tag{A.3}$$

Table 1: *Feedback mode generators* (see Section 5). Experiments show longer cycles than 2^{32}, but it is not guaranteed. If a linear sequence is added (+lin), the cycle length is at least 2^{32}.

Stages	Generator#	Cycles	Byte Ops	16-bit Ops	Special Ops	Invertible	Fails
1	(1)	5	1	—	Rot	Y	—
	(1)+lin	6	1	—	Rot	N	—
	(1)∗	9	2	—	—	Y	—
	(2)	9	3	—	—	N	—
	(3)	9	—	1	—	Y	1
	(4)	7	1	—	—	Y	7
	(4)+lin	8	1	—	—	N	—
	(5)	7	2	—	—	N	9
	(5)+lin	8	2	—	—	N	—
2	(1)	4	1	—	—	N	$1 \sim 2$
	(2)	4	All	—	—	N	12
	(2)+lin	5	All		—	N	—
	(3)	3	—	—	Rot	Y	—
	(4,xFib)	4	All	—	Rot	N	—
3	(1)	4	All	—	—	Y	—
	(1):[5,3]	4	—	—	—	Y	—
	(2)	4	All	—	—	Y	—
	(2):[5,3]	4	—	—	—	Y	—
	(3)	3	All	—	Rot	Y	—
	(4)	3	All	—	Rot	Y	—
	(5)	2	—	—	Rot	Y	—
	(6)	2	All	—	Rot	Y	—
	(7)	2	All	—	Rot	Y	—
4	(1)	2	All	—	Rot	Y	—
5	(2)	1	—	All	Swap	Y	—
	S6.5:(1)	1	All	—	BSWAP	Y	—

Table 2: *Counter mode generators* (see Section 6). The generators have the same cycle lengths as their internal counters: $2^{32\cdot \text{Stages}}$.

Stages	Generator#	Cycles	Byte Ops	16-bit Ops	Special Ops	Fails	Near fails
1	(1)	34	—	—	—	—	—
	(2)	59	All	—	—	—	—
	(3)	23	—	—	Rot	—	1
	(4)	43	5	5	Rot	—	2
2	(1)	25.5	—	—	—	—	—
	(2)	25.5	—	—	—	—	1
	(3)	25.5	—	—	—	—	—
	(4)	25.5	—	—	—	—	—
	(5)	31.5	11	—	—	—	1
	(6) $\cdots$ (10)	19.5	—	—	Rot	—	—
	(11)	29	10	10	Rot	—	2
4	(1) $\cdots$ (3)	18.5	—	—	—	—	—
	(4)	22.5	All	—	—	—	—
	(5), (6)	16	—	—	Rot	—	—
	(7)	18	All	—	Rot	—	—
	(8) $\cdots$ (11)	16	—	—	Rot	—	—
	(12) ...	18	All	—	Rot	—	—

Table 3: *Hybrid feedback mode generators* (see Section 7). These generators all use a 32-bit counter, thus their cycle lengths are at least 2^{32}, but experiments show much larger values.

Stages	Generator#	Cycles	Byte Ops	16-bit Ops	Special Ops	Fails	Near fails
2	(1)..	6	—	—	—	—	1
	(2)	8	—	—	—	—	—
	(3)	8	All	1	—	—	—
	(4)	4	—	1	Rot	—	—
	(5)	4	—	1	Rot	—	—
	(6)	6	All	1	Rot	—	—
	(7)	6	All	1	Rot	—	—
3	(1), (2)	5	All	—	—	—	—
	(3), (4)	3	All	—	Rot	—	—
	(5)	2	—	—	Rot	—	1
4	(1)	2	All	—	Rot	—	—
6	(1)	1	All	—	BSWAP	—	—

The recursion above defines the mixed Fibonacci generator. It provides another two-stage alternative to the Fibonacci generator ($x_{i+1} = x_i + x_{i-1}$). The LS bits are still too regular, so it is only good as a component in a compound generator. When started with small initial values, in each addition step the operand length increases by one, therefore they reach soon the full 32-bit length.

With initial values of $\{1, 2\}$, it provides $3 \cdot 2^{m-1}$ pseudorandom values ($3 \cdot 2^{m-2}$ long period) for word lengths $m > 4$ bits. It can be easily verified up to $m = 40$, and above.

NOTATIONS

$\oplus$ Exclusive or operation, XOR, the binary addition without carry. It is the same as binary polynomial addition, if the coefficients are represented by the bits of a machine word.

$\ll$ Shift left of an unsigned binary word, entering 0′s at the least significant position, discarding overflow bits. $3 \ll 2 = 12$.

$\gg$ Shift right of an unsigned binary word, entering 0′s at the most-significant position, discarding shifted out least-significant bits. $15 \gg 2 = 3$.

$\lll$ Rotate to the left. In program code we use functional notations: `rot(word,#bits)`, which is the same as `(word` $\ll$ `#bits) + (word` $\gg$ `(32- #bits))` at 32-bit machine words.

REFERENCES

[1] G. Marsaglia, "DIEHARD: a battery of tests of randomness," 1996, http://stat.fsu.edu/pub/diehard/.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, chapter 3, Addison-Wesley, Reading, Mass, USA, 2nd edition, 1981.

[3] G. Fishmann and L. R. Moore III, "An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$," *SIAM Journal of Scientific and Statistical Computing*, vol. 7, no. 1, pp. 24–45, 1985.

[4] P. L'Ecuyer, "Efficient and portable combined random number generators," *Communications of the ACM*, vol. 31, no. 6, pp. 742–751, 1988.

[5] F. James, "A review of pseudorandom number generators," in *Computer Physics Communication*, vol. 60, pp. 329–344, North Holland, Amsterdam, The Netherlands, 1990.

[6] M. Richter, *Ein Rauschgenerator zur Gewinnung von quasi-idealen Zufallszahlen fuer die stochastische Simulation*, Ph.D. thesis, Aachen University of Technology, Aachen, Germany, 1992.

[7] R. C. Tausworthe, "Random numbers generated by linear recurrence modulo two," *Mathematics of Computation*, vol. 19, no. 90, pp. 201–209, 1965.

[8] S. L. Anderson, "Random number generators on vector supercomputers and other advanced architectures," *SIAM Review*, vol. 32, no. 2, pp. 221–251, 1990.

[9] S. W. Golomb, *Shift Register Sequences*, Aegean Park Press, Walnut Creek, Calif, USA, Revised edition, 1982.

[10] G. Marsaglia, "A current view of random number generators," in *Computer Science and Statistics: The Interface*, L. Billard, Ed., pp. 3–10, Elsevier Science B.V., (North-Holland), Amsterdam, The Netherlands, 1985.

[11] M. Mascagni, S. Cuccaro, D. Pryor, and M. Robinson, "A fast, high quality, reproducible, parallel, lagged-Fibonacci pseudorandom number generator," Tech. Rep. SRC-TR-94-115, Supercomputing Research Center, 17100 Science Drive, Bowie, Md, USA, 1994.

[12] S. K. Park and K. W. Miller, "Random number generators: good ones are hard to find," *Communications of the ACM*, vol. 31, no. 10, pp. 1192–1201, 1988.

[13] D. Pryor, S. Cuccaro, M. Mascagni, and M. Robinson, "Implementation and usage of a portable and reproducible parallel pseudorandom number generator," Tech. Rep. SRC-TR-94-116, Supercomputing Research Center, 17100 Science Drive, Bowie, Md, USA, 1994.

[14] P. L'Ecuyer, "Maximally equidistributed combined Tausworthe generators," *Mathematics of Computation*, vol. 65, no. 213, pp. 203–213, 1996.

[15] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, "The RC6 Block Cipher," ftp://ftp.rsasecurity.com/pub/rsalabs/rc6/rc6v11.pdf.

[16] A. Klimov and A. Shamir, "A new class of invertible mappings," in *Proceedings of the 4th Workshop on Cryptographic Hardware and Embedded Systems (CHES '02)*, vol. 2523 of *Lecture Notes in Computer Science*, pp. 471–484, Redwood Shores, Calif, USA, August 2002.

[17] A. Biryukov and D. Wagner, "Slide attacks," in *Proceedings of the 6th International Workshop on Fast Software Encryption (FSE '99)*, L. Knudsen, Ed., vol. 1636 of *Lecture Notes In Computer Science*, pp. 245–259, Rome, Italy, March 1999.

[18] M. D. Russell, "Tinyness: An Overview of TEA and Related Ciphers," http://www-users.cs.york.ac.uk/matthew/TEA/.

[19] D. J. Wheeler and R. M. Needham, "TEA, a tiny encryption algorithm," in *Proceedings of the 6th International Workshop on Fast Software Encryption (FSE '94)*, B. Preneel, Ed., vol. 1008 of *Lecture Notes in Computer Science*, pp. 363–366, Leuven, Belgium, December 1994.

[20] D. J. Wheeler and R. M. Needham, "Correction to XTEA," Tech. Rep., Computer Laboratory, University of Cambridge, Cambridge, UK, October 1998.

[21] S. Even and Y. Mansour, "A construction of a cipher from a single pseudorandom permutation," in *Advances in Cryptology - ASIACRYPT '91, Proceedings of International Conference on the Theory and Applications of Cryptology*, vol. 739 of *Lecture Notes in Computer Science*, pp. 210–224, Fujiyoshida, Japan, November 1991.

[22] I. B. Damgård, "A design principle for hash functions," in *Advances in Cryptology - CRYPTO '89, Proceedings of the 9th Annual International Cryptology Conference*, vol. 435 of *Lecture Notes in Computer Science*, pp. 416–427, Santa Barbara, Calif, USA, August 1989.

[23] R. Merkle, "One-way hash functions and DES," in *Advances in Cryptology - CRYPTO '89, Proceedings of the 9th Annual International Cryptology Conference*, vol. 435 of *Lecture Notes in Computer Science*, pp. 428–446, Santa Barbara, Calif, USA, August 1989.

[24] Davies-Meyer, "Double-Pipe Hash construction and their combinations".

[25] S. Lucks, "Design Principles for Iterated Hash Functions," IACR eprint archive, September 2004, http://eprint.iacr.org/2004/253.pdf.

[26] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, Fla, USA, 1996.