広島大学学位請求論文

A fast pseudorandom number generator and a fast stream cipher
（高速擬似乱数生成器および高速ストリーム暗号）

２００９年

広島大学大学院理学研究科
数学専攻

齋藤　睦夫

（広島大学大学院理学研究科助教）

目　次

主論文

A fast pseudorandom number generator and a fast stream cipher

Mutsuo Saito

Introduction

In this thesis, we describe two topics about pseudorandom number generators (PRNGs). The first topic is a PRNG itself and the second is a stream cipher.

One of the biggest applications of PRNGs is Monte Carlo Simulations. The advances of computers have been making the scale of scientific simulations larger and larger. Such a simulation often requires high speed and high quality PRNGs.

For the last decade, Mersenne Twister (MT)[21] has been widely used for scientific simulations and has gotten high respect. In Chapter 1, we propose a new PRNG which is much faster and has better quality of randomness than MT.

The new PRNG is based on a Single Instruction Multiple Data[30] (SIMD) feature of recent CPUs. We call it SIMD-oriented Fast Mersenne Twister (SFMT). SFMT generates a sequence of 128-bit integers, and use each of them as four 32-bit integers or two 64-bit integers. We calculate the dimension of equidistribution which shows one property of randomness of SFMT, a criteria for pseudorandomness.

Chapter 1 is a joint work with Makoto Matsumoto. This content is written in [29].

Chapter 2 treats a stream cipher. Stream cipher is a kind of encoding method of cryptography, and widely used in Internet technology. As commercial use of Internet is getting larger, the importance of stream cipher is getting larger.

Stream cipher is a kind of PRNG, and it needs to satisfy extra request that the outputs are difficult to be predicted. In many cases, stream ciphers are designed using non-linearity, and it is difficult to predict the output but often difficult to analyze their behavior theoretically.

We prove a theorem that assures the period and the equidistribution property of PRNG with a quasi-group filter. We implement an instance called CryptMT.

CryptMT ver.3 use a variant of SFMT as its mother generator, that means, mother generator is 128-bit based, and its filter is also 128-bit SIMD based. We show CryptMT ver.3 is very fast, especially in core 2, the newest CPU of Intel, with theoretically assured long period.

Chapter 2 is a joint work with Makoto Matsumoto, Takuji Nishimura, and Mariko Hagita. The content is published in [26].

Chapter 1

A fast pseudorandom number generator, SFMT

1 Introduction

Computer Simulation is one of the most important techniques of modern science. Recently, the scale of simulations is getting larger, and faster pseudorandom number generators (PRNGs) are required. Power of CPUs for usual personal computers is now sufficiently strong for such purposes, and necessity of efficient PRNGs for CPUs on PCs is increasing. One such generator is Mersenne Twister (MT) [21], which is based on a linear recursion modulo 2 over 32-bit words. An implementation MT19937 has the period of $2^{19937} - 1$.

There is an argument that the CPU time consumed for function calls to PRNG routines occupies a large part of the random number generation, and consequently it is not so important to improve the speed of PRNG (cf. [27]). This is not always the case: one can avoid the function calls by (1) inline-expansion and/or (2) generation of pseudorandom numbers in an array at one call.

Our aim of this chapter is to design a fast MT-like PRNG (i.e. Linear Feedbacked Shift Register) considering new features of modern CPUs on PCs.

1.1 Linear Feedbacked Shift Register (LFSR) generators

An LFSR method is to generate a sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ of elements $\mathbb{F}_2^w$ by a recursion

$$\mathbf{x}_{i+N} := g(\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{i+N-1}), \tag{1}$$

where $\mathbf{x}_i \in \mathbb{F}_2^w$ and $g : (\mathbb{F}_2^w)^N \to \mathbb{F}_2^w$ is an $\mathbb{F}_2$-linear function (i.e., the multiplication of a $(wN \times w)$-matrix from the right to a wN-dimensional vector) and use it as a pseudorandom w-bit integer sequence. In the implementation, this recursion is computed by using an array `W[0..N-1]` of N integers of w-bit size, by the simultaneous substitutions

$$\texttt{W[0]} \leftarrow \texttt{W[1]},\ \texttt{W[1]} \leftarrow \texttt{W[2]},\ \ldots, \texttt{W[N-2]} \leftarrow \texttt{W[N-1]},\ \texttt{W[N-1]} \leftarrow g(\texttt{W[0]}, \ldots, \texttt{W[N-1]}).$$

The first $N - 1$ substitutions shift the content of the array, hence the name of LFSR. Note that in the implementation we may use an indexing technique to avoid computing these substitutions, see [15, P.28 Algorithm A]. The array `W[0..N-1]` is called the state array. Before starting the generation, we need to set some values to the state array, which is called the initialization.

Mersenne Twister (MT) [21] is an example with

$$g(\mathbf{w}_0, \ldots, \mathbf{w}_{N-1}) = (\mathbf{w}_0|\mathbf{w}_1)A + \mathbf{w}_M,$$

where $(\mathbf{w}_0|\mathbf{w}_1)$ denotes the concatenation of the $32 - r$ most significant bits (MSBs) of $\mathbf{w}_0$ and the r least significant bits (LSBs) of $\mathbf{w}_1$, A is a (32×32)-matrix for which the multiplication $\mathbf{w}A$ is computable by a few bit-operations, and M is an integer ($1 < M < N$). Its period is $2^{32N-r} - 1$, chosen to be a Mersenne prime. To obtain a better equidistribution property, MT transforms the sequence by a suitably chosen (32×32) matrix T, namely, MT outputs $\mathbf{x}_0 T, \mathbf{x}_1 T, \mathbf{x}_2 T, \ldots$ (called tempering).

An advantage of $\mathbb{F}_2$-linear generators over integer multiplication generators (such as Linear Congruential Generators [15] or Multiple Recursive Generators [16]) was high-speed generation by avoiding multiplications. Another advantage is that the behavior of generated pseudorandom number sequence is theoretically well studied and its dimension of equidistribution can be calculated easily.

1.2 Single Instruction Multiple Data (SIMD)

Single Instruction Multiple Data (SIMD) [30] is a technique employed to achieve data level parallelism. Typically, four 32-bit registers are combined into a 128-bit register, and a single instruction operates on the 128-bit register. There are two types of SIMD instructions. One is to operate four 32-bit registers separately (e.g. addition and subtraction.) The other is to operate on the 128-bit integer (e.g. 128-bit shift operation.)

SIMD is also called Multimedia Extension because main target applications of SIMD are multimedia applications, which use huge data like image or sound. LFSR uses a large internal state array, so SIMD can be used to accelerate its generation.

Streaming SIMD Extensions 2 (SSE2) [14, Chapter 4–5] is one of the SIMD instruction sets introduced by Intel. Pentium M, Pentium 4 and later CPUs support SSE2, but Itanium and Itanium 2 do not. AMD Athlon 64, Opteron and Turion 64 also support SSE2. These CPUs have eight 128-bit registers and each register can be divided into 8-bit, 16-bit, 32-bit or 64-bit blocks.

AltiVec [12] is another SIMD instruction-set supported by PowerPC G4 and G5. These CPUs have thirty-two 128-bit registers and each register can be divided into 8-bit, 16-bit or 32-bit blocks.

Our purpose is to design a PRNG which can be implemented efficiently both in SSE2 and AltiVec.

Intel C compiler has an ability to handle SSE2 instructions. GCC C compiler, which is more widely used, also has macros and inline functions to handle SSE2 and AltiVec instructions directly.

2 SIMD-oriented Fast Mersenne Twister

In this chapter, we propose SIMD-oriented Fast Mersenne Twister (SFMT) pseudorandom number generators. They are LFSR generators based on a recursion over $\mathbb{F}_2^{128}$. We identify the set of bits $\{0, 1\}$ with the two element field $\mathbb{F}_2$. This means that every arithmetic operation is done modulo 2. A w-bit integer is identified with a horizontal vector in $\mathbb{F}_2^w$, and $+$ denotes the sum as vectors (i.e., bit-wise exor), not as integers. We consider three cases: w is 32, 64 or 128.

2.1 Memory and period

Long period is desirable for PRNGs, however, long period generator needs large memory. This is a primary trade-off of PRNGs. We decided to make a set of generators, each of them has different period and different memory size. Table 1 shows memories and periods of SFMT. The column MEXP means the Mersenne exponent: namely, $2^{\mathrm{MEXP}}-1$ is a prime (called a Mersenne Prime). We obtained parameters for SFMT with period $2^{\mathrm{MEXP}}-1$, for $607 \geq \mathrm{MEXP} \geq 216091$. For the minimum MEXP 607, SFMT consumes about 80 byte and has a period of at least $2^{607}-1 \simeq 5.31 \times 10^{182}$. For the maximum MEXP 216091, SFMT consumes about 26K byte and has a period of at least $2^{216091}-1 \simeq 7.46 \times 10^{65049}$.

In this chapter we mainly describe about SFMT19937 (i.e. a particular SFMT with MEXP = 19937), however, what we describe applies to SFMT with any MEXP.

Table 1: memories and periods

MEXP	size of array	memory	least period
607	5	80 byte	$\simeq 5.31 \times 10^{182}$
1279	10	160 byte	$\simeq 1.04 \times 10^{385}$
2281	18	288 byte	$\simeq 4.46 \times 10^{686}$
4253	34	544 byte	$\simeq 1.90 \times 10^{1280}$
11213	88	1,408 byte	$\simeq 2.81 \times 10^{3375}$
19937	156	2,496 byte	$\simeq 4.32 \times 10^{6001}$
44497	348	5,568 byte	$\simeq 8.55 \times 10^{13394}$
86243	674	10,784 byte	$\simeq 5.37 \times 10^{25961}$
132049	1032	16,512 byte	$\simeq 5.13 \times 10^{39750}$
216091	1689	27,024 byte	$\simeq 7.46 \times 10^{65049}$

2.2 The recursion of SFMT

We choose g in the recursion (1) as

$$g(\mathbf{w}_0, \ldots, \mathbf{w}_{N-1}) = \mathbf{w}_0 A + \mathbf{w}_M B + \mathbf{w}_{N-2} C + \mathbf{w}_{N-1} D, \qquad (2)$$

where $\mathbf{w}_0, \mathbf{w}_M, \ldots$ are $w(=128)$-bit integers (= horizontal vectors in $\mathbb{F}_2^{128}$), and A, B, C, D are sparse 128×128 matrices over $\mathbb{F}_2$ for which $\mathbf{w}A, \mathbf{w}B, \mathbf{w}C, \mathbf{w}D$ can be computed by a few SIMD bit-operations. The choice of the suffixes $N-1$, $N-2$ is for speed: in the implementation of g, `W[0]` and `W[M]` are read from the array `W`, while the copies of `W[N-2]` and `W[N-1]` are kept in two 128-bit registers in the CPU, say `r1` and `r2`. Concretely speaking, we assign `r2` $\leftarrow$ `r1` and `r1` $\leftarrow$ "the result of (2)" at every generation, then `r2` (`r1`) keeps a copy of `W[N-2]` (`W[N-1]`, respectively). The merit of doing this is to use the pipeline effectively. To fetch `W[0]` and `W[M]` from memory takes some time. In the meantime, the CPU can compute $\mathbf{w}_{N-2}C$ and $\mathbf{w}_{N-1}D$, because copies of $\mathbf{w}_{N-2}$ and $\mathbf{w}_{N-1}$ are kept in the registers. This selection was made through experiments on the speed of generation.

By trial and error, we searched for a set of parameters of SFMT, with the period being a multiple of $2^{19937}-1$ and having good equidistribution properties. The degree of recursion N is $\lceil 19937/128 \rceil = 156$, and the linear transformations A, B, C, D are as follows.

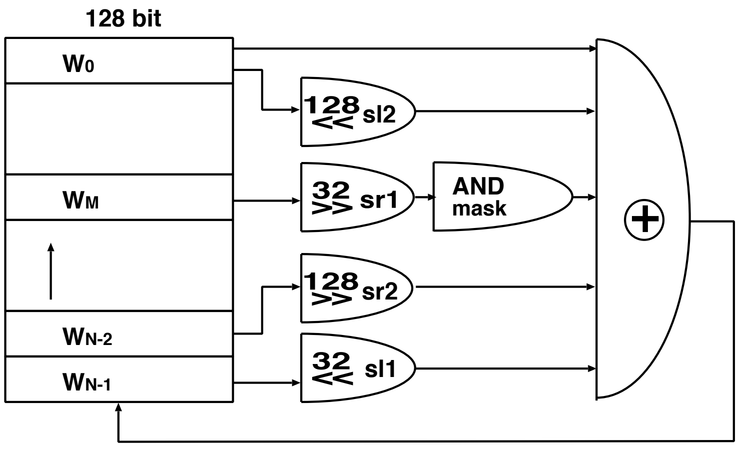

Figure 1: A circuit-like description of SFMT19937.

- $\mathbf{w}A := (\mathbf{w} \overset{128}{<<} \mathit{sl2}) + \mathbf{w}$.

  This notation means that $\mathbf{w}$ is regarded as a single 128-bit integer, and $\mathbf{w}A$ is the result of the left-shift of $\mathbf{w}$ by *sl2* bits. There is such a SIMD operation in both Pentium SSE2 and PowerPC AltiVec SIMD instruction sets (SSE2 permits only a multiple of 8 as the amount of shifting). Note that the notation $+$ means the exclusive-or in this thesis.

- $\mathbf{w}B := (\mathbf{w} \overset{32}{>>} \mathit{sr1})\&(\mathit{mask})$.

  This notation means that $\mathbf{w}$ is considered to be a quadruple of 32-bit integers, and each 32-bit integer is shifted to the right by *sr1* bits, (thus the eleven most significant bits are filled with 0s, for each 32-bit integer). The C-like notation & means the bitwise AND with a constant 128-bit mask.

  In the search, this constant is generated as follows. Each bit in the 128-bit integer is independently randomly chosen, with the probability to choose 1 being 7/8. This is because we prefer to have more 1's for a denser feedback.

- $\mathbf{w}C := (\mathbf{w} \overset{128}{>>} \mathit{sr2})$.

  This is the right shift of a 128-bit integer by 8 bits, similar to the first.

- $\mathbf{w}D := (\mathbf{w} \overset{32}{<<} \mathit{sl1})$.

  Similar to the second, $\mathbf{w}$ is cut into four pieces of 32-bit integers, and each of these is shifted by 18 bits to the left.

All these instructions are available in both Intel Pentium's SSE2 and PowerPC's AltiVec SIMD instruction sets. Figure 1 shows a concrete description of SFMT19937 generator with period a multiple of $2^{19937} - 1$. Table 2 shows the parameter sets for all MEXPs.

Table 2: parameter sets

MEXP	M	$sl1$	$sl2$	$sr1$	$sr2$	MASK
607	2	15	24	13	24	`7FF7FB2F FF777B7D EF7F3F7D FDFF37FF`
1279	3	18	8	1	8	`FFFFFD7D B3FDAFF9 37F5EFFB FFFFFFFB`
2281	12	19	8	5	8	`F2F7CBBF F7FFEF7F FDFFFFFE BFF7FFBF`
4253	17	20	8	7	8	`FFFFF7BB 3EFFFFFB 9FFFFF5F 9F7BFFFF`
11213	68	14	24	7	24	`7FFFDBFD DFDFBFFF FFFFFFEF EFFFF7FB`
19937	122	18	8	11	8	`BFFFFFF6 BFFAFFFF DDFECB7F DFFFFFEF`
44497	330	5	24	9	24	`9FFD7BFF BFBF7BEF DFBEBFFF EFFFFFFB`
86243	366	6	56	19	8	`BF9FF3FF FD77EFFF BFF7FF3F FDBFFBFF`
132049	110	19	8	21	8	`CFF77FFF FFFEFFFA FB6EBF95 FFFFBB5F`
216091	627	11	24	10	8	`FFDDFBFB BFFFFA7F BFFFFFFF BFF7BFF7`

This table shows parameter sets of SFMT for MEXPs from 607 to 216091. The parameters $sl2$ and $sr2$ are selected to be multiple of 8.

2.3 Endianness

Let $\mathbf{x}[0..3]$ be an array of 32-bit integers of size four. There are two natural ways to convert the array to a 128-bit integer. One is to concatenate in the order of $\mathbf{x}[3]\mathbf{x}[2]\mathbf{x}[1]\mathbf{x}[0]$, from MSBs to LSBs, which is called the little-endian system, adopted in Pentium. The converse is the big-endian system adopted in PowerPC, see [10].

The descriptions in this thesis is based on the former. To assure the portability for both endian systems, we implemented two codes: one is for little-endian system (SSE2 of Pentium) and the other is for big-endian system (AltiVec of PowerPC), to assure the exactly same outputs as 32-bit integer generators. In the latter code, the recursion (2) is considered as a recursion on quadruples of 32-bit integers, rather than 128-bit integers, so that the content of the state array coincides both for little and big endian systems, as an array of 32-bit integers (not as 128-bit integers). Thus, shift-operations on 128-bit integers in the little-endian system is different from that in the big-endian system. PowerPC supports arbitrary permutations of 16 blocks of 8-bit integers in a 128-bit register, which can emulate the shift in (2).

2.4 Block-generation

In the block-generation scheme, the user of the PRNG specifies an array of w-bit integers of the length L, where $w = 32$, 64 or 128 and L is specified by the user. In the case of SFMT19937, wL should be a multiple of 128 and no less than $N \times 128$, since the array needs to accommodate the state space (note that $N = 156$). By calling the block generation function with the pointer to this array, w, and L, the routine fills up the array with pseudorandom integers, as follows. SFMT19937 keeps the state space S in an internal array of 128-bit integers of length 156. We concatenate this state array with the user-specified array, using the indexing technique. Then, the routine generates 128-bit integers in the user-specified array by recursion (2), as described in Figure 2, until it fills up the array. The last 156 128-bit integers are copied back to the internal array of SFMT19937. This makes the generation much faster than sequential generation (i.e., one generation per one call) as shown in Table 5.

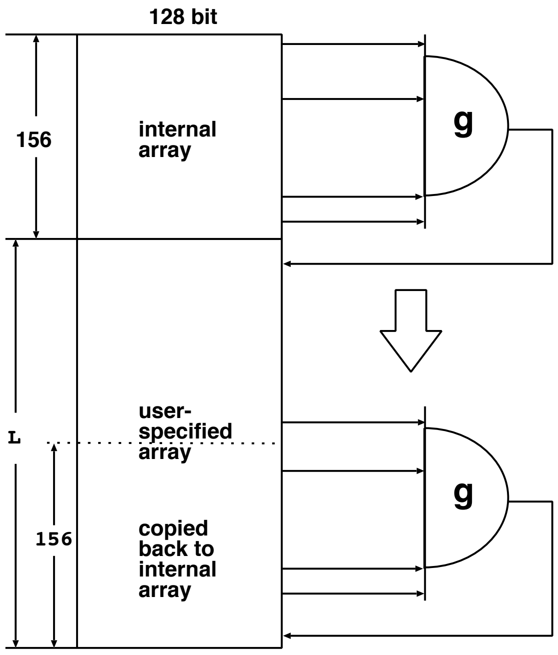

Figure 2: Block-generation scheme

3 How to select the recursion and parameters.

We wrote a code to compute the period and the dimensions of equidistribution (DE, see §3.2). Then, we searched for a recursion with good DE admitting a fast implementation.

3.1 Computation of the Period

An LFSR that obeys the recursion (1) may be considered as an automaton, with the state space $S = (\mathbb{F}_2^w)^N$ and the state transition function $f : S \to S$ given by $(\mathbf{w}_0, \dots, \mathbf{w}_{N-1}) \mapsto (\mathbf{w}_1, \dots, \mathbf{w}_{N-1}, g(\mathbf{w}_0, \dots, \mathbf{w}_{N-1}))$. As a w-bit integer generator, the output function is $o : S \to \mathbb{F}_2^w$, $(\mathbf{w}_0, \dots, \mathbf{w}_{N-1}) \mapsto \mathbf{w}_0$.

Let χ_f be the characteristic polynomial of $f : S \to S$. If χ_f is primitive, then the period of the state transition takes the maximal value $2^{\dim(S)} - 1$ [15, §3.2.2]. However, to check the primitivity, we need the integer factorization of this number, which is often hard for $\dim(S) = nw > 10000$. On the other hand, the primarity test is much easier than the factorization, so many huge primes of the form $2^p - 1$ have been found. Such a prime is called a Mersenne prime, and p is called the Mersenne exponent, which itself is a prime.

MT and WELL[28] discard r specific bits from the array S, so that $nw - r$ is a Mersenne exponent. Then, the primitivity of χ_f is easily checked by the algorithm in [15, §3.2.2], avoiding the integer factorization.

SFMT adopted another method to avoid the integer factorization, the reducible transition method (RTM), which uses a reducible characteristic polynomial with a large primitive factor. This idea appeared in [13] [3][4]. We briefly recall it.

Let p be the Mersenne exponent, and $N := \lceil p/w \rceil$. Then, we randomly choose parameters for the recursion of LFSR (1). By applying the Berlekamp-Massey Algorithm to the output sequence, we obtain $\chi_f(t)$. (Note that a direct computation of $\det(tI - f)$ is time-consuming because $\dim(S) = 19968$.)

By using a sieve, we remove all factors of small degree from χ_f, until we know that it has no irreducible factor of degree p, or that it has a (possibly reducible) factor of degree p. In the latter case, the factor is passed to the primitivity test described in [15, §3.2.2].

Suppose that we found a recursion with an irreducible factor of desired degree p in $\chi_f(t)$. Then, we have a factorization

$$\chi_f = \phi_p \phi_r,$$

where ϕ_p is a primitive polynomial of degree p and ϕ_r is a polynomial of degree $r = wN - p$. These are coprime, since we assume $p > r$. Let $\mathrm{Ker}(g)$ denote the kernel of a linear transformation g. By putting $V_p := \mathrm{Ker}\,(\phi_p(f))$ and $V_r := \mathrm{Ker}\,(\phi_r(f))$, we have a decomposition into f-invariant subspaces

$$S = V_p \oplus V_r \quad (\dim V_p = p,\ \dim V_r = r).$$

Note that the characteristic polynomial of the restriction f_p of f to V_p is $\phi_p(t)$, and that of the restriction f_r to V_r is $\phi_r(t)$. For any state $s \in S$, we denote $s = s_p + s_r$ for the corresponding decomposition with $s_p \in V_p$ and $s_r \in V_r$. Then, the k-th state $f^k(s)$ is equal to $f_p^k(s_p) + f_r^k(s_r)$. This implies that the automaton is equivalent to the sum of two automata $f_p : V_p \to V_p$ and $f_r : V_r \to V_r$. To combine two linear automata by sum is well-studied as combined Tausworthe generators or combined LFSRs, see [7] [17] [18]. Their purpose is to obtain a good PRNG from several simple generators, which is different from ours.

The period length of the state transition is the least common multiple of that started from s_p and that started from s_r. Hence, if $s_p \neq 0$, then the period is a nonzero multiple of $2^p - 1$.

To assure that $s_p \neq 0$, we look some number of bits in the state space S more than $r = \dim V_r$ bits. For example, we pick one of `W[i]`. By looking at only the bits in `W[0]`, we have a projection $\pi : S \to V_{W_0}$, where we assume $\dim V_{W_0} > r$. Now the image $\pi(V_r) \subset V_{W_0}$ is a proper subspace. If we equip the standard inner product in V_{W_0}, then there is a nonzero vector which is orthogonal to $\pi(V_r)$. We call such a vector *the period certification vector* (PCV). Such vector can be obtained by solving a linear equation, representing the orthogonality to all $\phi_p(f)(b)$ for b moves in a fixed basis of S.

The inner product of PCV and any vector in $\pi(V_r)$ is zero. Hence, if a state $s \in S$ has non-zero inner product $\mathrm{PCV} \cdot \pi(s) = 1$, then it implies that $s \notin V_r$, and thus $s_p \neq 0$, which assures the period at least $2^p - 1$ for f. Note that this inner product is taken in V_{W_0}, i.e., looking at the bits in `W[0]` of s. If the inner product happens to be 0, then change one bit in $\pi(s)$, where PCV has 1, so that the inner product becomes one. In sum, the initial state can not be chosen arbitrarily: one bit in `W[0]` is computed from the rest bit in `W[0]`, to assure the period being a multiple of $2^p - 1$. Table 3 lists the period certification vectors(PCV).

Remark 3.1. *The number of non-zero terms in $\chi_f(t)$ is an index measuring the amount of bit-mixing. In the case of SFMT19937, the number of nonzero*

Table 3: period certification vectors

MEXP	PCV1	PCV2	PCV3	PCV4
607	`0x00000001`	`0x00000000`	`0x00000000`	`0x5986F054`
1279	`0x00000001`	`0x00000000`	`0x00000000`	`0x20000000`
2281	`0x00000001`	`0x00000000`	`0x00000000`	`0x41DFA600`
4253	`0xa8000001`	`0xAF5390A3`	`0xB740B3F8`	`0x6C11486D`
11213	`0x00000001`	`0x00000000`	`0xE8148000`	`0xD0C7AFA3`
19937	`0x00000001`	`0x00000000`	`0x00000000`	`0x13C9E684`
44497	`0x00000001`	`0x00000000`	`0xA3AC4000`	`0xECC1327A`
86243	`0x00000001`	`0x00000000`	`0x00000000`	`0xE9528D85`
132049	`0x00000001`	`0x00000000`	`0xCB520000`	`0xC7E91C7D`
216091	`0xF8000001`	`0x89E80709`	`0x3BD2B64B`	`0x0C64B1E4`

terms is 6711, which is much larger than 135 of MT, but smaller than 8585 of WELL19937c [28]. Table 4 shows number of non-zero terms in $\chi_f(t)$ for all MEXPs of SFMT.

Table 4: number of non-zero terms in $\chi_f(t)$

MEXP	degree of $\chi_f(t)$	non-zero term	ratio
607	640	288	0.45
1279	1280	582	0.45
2281	2304	966	0.42
4253	4352	1830	0.42
11213	11264	4266	0.38
19937	19968	6711	0.34
44497	44544	12484	0.28
86243	86272	7069	0.08
132049	132096	18158	0.14
216091	216192	6826	0.03

3.2 Computation of the dimension of equidistribution

We recall the definition of dimension of equidistribution (cf. [7][17]).

Definition 3.2. *A periodic sequence with period P*

$$\chi := \mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{P-1}, \mathbf{x}_P = \mathbf{x}_0, \ldots$$

*of v-bit integers is said to be k-*dimensionally equidistributed *if any kv-bit pattern occurs equally often as a k-tuple*

$$(\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{i+k-1})$$

for a period $i = 0, \ldots, P-1$. We allow an exception for the all-zero pattern, which may occur once less often. (This last loosening of the condition is technically necessary, because the zero state does not occur in an $\mathbb{F}_2$-linear generator). The largest value of such k is called the dimension of equidistribution (DE).

We want to generalize this definition slightly. We define the k-window set of the periodic sequence χ as

$$W_k(\chi) := \{(\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{i+k-1}) | i = 0, 1, \ldots, P-1\},$$

which is considered as a *multi-set*, namely, the multiplicity of each element is considered.

For a positive integer m and a multi-set T, let us denote by $m \cdot T$ the multi-set where the multiplicity of each element in T is multiplied by m. Then, the above definition of equidistribution is equivalent to

$$W_k(\chi) = (m \cdot \mathbb{F}_2^{vk}) \setminus \{\mathbf{0}\},$$

where m is the multiplicity of the occurrences, and the operator $\setminus$ means that the multiplicity of $\mathbf{0}$ is subtracted by one.

Definition 3.3. *In the above setting, if there exist a positive integer m and a multi-subset $D \subset (m \cdot \mathbb{F}_2^{vk})$ such that*

$$W_k(\chi) = (m \cdot \mathbb{F}_2^{vk}) \setminus D,$$

we say that χ is k-dimensionally equidistributed with defect ratio $\#(D)/\#(m \cdot \mathbb{F}_2^{vk})$, where the cardinality is counted with multiplicity.

Thus, in Definition 3.2, the defect ratio up to $1/(P+1)$ is allowed to claim the dimension of equidistribution. If $P = 2^{19937} - 1$, then $1/(P+1) = 2^{-19937}$. In the following, the dimension of equidistribution allows the defect ratio up to 2^{-19937}.

For a w-bit integer sequence, its *dimension of equidistribution at v-bit accuracy $k(v)$* is defined as the DE of the v-bit sequence, obtained by extracting the v MSBs from each of the w-bit integers. If the defect ratio is $1/(P+1)$, then there is an upper bound

$$k(v) \le \lfloor \log_2(P+1)/v \rfloor.$$

The gap between the realized $k(v)$ and the upper bound is called the dimension defect at v of the sequence, and denoted by

$$d(v) := \lfloor \log_2(P+1)/v \rfloor - k(v).$$

The summation of all the dimension defects at $1 \le v \le 32$ is called the total dimension defect, denoted by Δ.

There is a difficulty in computing $k(v)$ when a 128-bit integer generator is used as a 32-bit (or 64-bit) integer generator. SFMT generates a sequence $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ of 128-bit integers. Then, they are converted to a sequence of 32-bit integers $\mathbf{x}_0[0], \mathbf{x}_0[1], \mathbf{x}_0[2], \mathbf{x}_0[3], \mathbf{x}_1[0], \mathbf{x}_1[1], \ldots$, where $\mathbf{x}[0]$ is the 32 LSBs of $\mathbf{x}$, $\mathbf{x}[1]$ is the 33rd–64th bits, $\mathbf{x}[2]$ is the 65rd–96th bits, and $\mathbf{x}[3]$ is the 32 MSBs. (This is called the little-endian system, see [10], for the notion of endianness, and §7 for an implementation in a big-endian system).

Then, we need to modify the model automaton as follows. The state space is $S' := S \times \{0, 1, 2, 3\}$, the state transition function $f' : S' \to S'$ is

$$f'(s, i) := \begin{cases} (s, i+1) & (\text{ if } i < 3), \\ (f(s), 0) & (\text{ if } i = 3) \end{cases}$$

and the output function is

$$o' : S' \to \mathbb{F}_2^{32},\ ((\mathbf{w}_0, \dots, \mathbf{w}_{N-1}), i) \mapsto \mathbf{w}_0[i].$$

We fix $1 \le v \le w$, and let $o_k(s, i)$ be the k-tuple of the v MSBs of the consecutive k-outputs from the state (s, i).

Proposition 3.4. *Assume that f is bijective. Let $k' = k'(v)$ denote the maximum k such that*

$$o_k(-, i) : V_p \to \mathbb{F}_2^{kv}, \quad s \mapsto o_k(s, i) \tag{3}$$

are surjective for all $i = 0, 1, 2, 3$. Take an initial state s satisfying $s_p \neq 0$. Then, the 32-bit output sequence is at least $k'(v)$-dimensionally equidistributed with v-bit accuracy with defect ratio 2^{-p}.

Moreover, if $4 < k'(v) + 1$, then for any initial state with $s = s_p \neq 0$ (hence $s_r = 0$), the dimension of equidistribution with defect ratio 2^{-p} is exactly $k'(v)$.

Proof. Take $s \in S$ with $s_p \neq 0$. Then, the orbit of s by f has the form of $(V_p - \{0\}) \times U \subset V_p \times V_r$, since $p > r$ and $2^p - 1$ is a prime. The surjectivity of the linear mapping $o_{k'}(-, i)$ implies that the image of

$$o_{k'}(-, i) : V_p \times U \to \mathbb{F}_2^{kv}$$

is $m \cdot \mathbb{F}_2^{kv}$ as a multi-set for some m. The defect comes from $0 \in V_p$, whose ratio in V_p is 2^{-p}. Then the first statement follows, since $W_{k'}(\chi)$ is the union of the images $o_{k'}(-, i)((V_p - \{0\}) \times U)$ for $i = 0, 1, 2, 3$.

For the latter half, we define L_i as the multiset of the image of $o_{k'+1}(-, i) : V_p \to \mathbb{F}_2^{(k'+1)v}$. Because of $s_r = 0$, we have $U = \{0\}$, and the union of $(L_i - \{0\})$ $(i = 0, 1, 2, 3)$ as a multi-set is $W_{k'+1}(\chi)$. If the sequence is $(k'+1)$-dimensionally equidistributed, then the multiplicity of each element in $W_{k'+1}(\chi)$ is at most $2^p \times 4/2^{(k'+1)v}$.

On the other hand, the multiplicity of an element in L_i is equal to the cardinality of the kernel of $o_{k'+1}(-, i)$. Let d_i be its dimension. Then by the dimension theorem, we have $d_i \ge p - (k' + 1)v$, and the equality holds if and only if $o_{k'+1}(-, i)$ is surjective. Thus, if there is a nonzero element $x \in \cap_{i=0}^3 L_i$, then its multiplicity in $W_{k'+1}(\chi)$ is no less than $4 \times 2^{p-(k'+1)v}$, and since one of $o_{k'+1}(-, i)$ is not surjective by the definition of k', its multiplicity actually exceeds $4 \times 2^{p-(k'+1)v}$, which implies that the sequence is not $(k'+1)$-dimensionally equidistributed, and the proposition follows. Since the codimension of L_i is at most v, that of $\cap_{i=0}^3 L_i$ is at most $4v$. The assumed inequality on k' implies the existence of nonzero element in the intersection. □

The dimension of equidistribution $k(v)$ depends on the choice of the initial state s. The above proposition implies that $k'(v)$ coincides with $k(v)$ for the worst choice of s under the condition $s_p \neq 0$. Thus, we adopt the following definition (analogously to t_l in [17]).

Definition 3.5. *Let k be the maximum such that (3) is satisfied. We call this the dimension of equidistribution of v-bit accuracy, and denote it simply by $k(v)$. We have an upper bound $k(v) \le \lfloor p/v \rfloor$.*

We define the dimension defect at v by

$$d(v) := \lfloor p/v \rfloor - k(v) \text{ and } \Delta := \sum_{v=1}^{w} d(v).$$

We may compute $k(v)$ by standard linear algebra. We used a more efficient algorithm based on a weighted norm, generalizing [7].

Here we briefly recall the method in [7]. Let us denote the output v-bit sequence from an initial state s_0 by

$$\begin{aligned} & b_{ij} \in \mathbb{F}_2 \\ o(s_0) &= (b_{10}, b_{20}, ..., b_{v0}), \\ o(s_1) &= (b_{11}, b_{21}, ..., b_{v1}), \\ & \vdots \end{aligned}$$

We assign to the initial state $s_o \in S$ a v-dimensional vector with components in the formal power series $A = \mathbb{F}_2[[t]]$:

$$w(s_0) = \left(\sum_{j=0}^{\infty} b_{1j} t^j, \sum_{j=0}^{\infty} b_{2j} t^j, ..., \sum_{j=0}^{\infty} b_{vj} t^j, \right).$$

This assignment $w : S \to F^v$ is an $\mathbb{F}_2$-linear function.

We consider the formal Laurent series field $F = \mathbb{F}_2((t)) \supset A$, and define its norm and the norm on F^v by

$$\begin{aligned} \left| \sum_{i=-m}^{\infty} a_i t^i \right| &:= 2^m \quad (a_{-m} \neq 0), \quad |0| = 0 \\ ||(x_1, ...x_v)|| &:= \max_{i=1,2,...,v} \{|x_i|\}. \end{aligned}$$

The polynomial ring $\mathbb{F}_2[t^{-1}]$ is discrete in F, and consider an $\mathbb{F}_2[t^{-1}]$-lattice $L \subset F^v$ defined by

$$\begin{aligned} e_i &:= (0, ..., 0, t^{-1}, 0, ..., 0) \quad \text{the } i\text{-th component is } t^{-1}, \text{ others being } 0 \\ L &:= \mathbb{F}_2[t^{-1}]\langle e_1, ..., e_v, w(s) \rangle \end{aligned}$$

Basic theorems used in [7] are the following.

Theorem 3.6. *Suppose that the PRNG satisfies the maximal period condition. If the covering radius of L is 2^{-k-1}, then the dimension of the equidistribution of the output v-bit sequence is k.*

Theorem 3.7. *The covering radius of L is 2^{-k-1} if and only if the shortest basis of L has the norm 2^{-k}.*

We may apply this method to SFMT19937, but only as a 128-bit integer generator, since SFMT19937 is not an $\mathbb{F}_2$-linear generator as 32 or 64-bit integer generator. Instead, we need to check the surjectivity of

$$o_k(-, i) : V_p \to \mathbb{F}_2^{kv}$$

for $i = 0, 1, 2, 3$. For simplicity we treat only the case $i = 0$, since other cases follow similarly.

Let x_j be the j-th output 128-bit integer of SFMT19937, and let $b_{j,m} \in \mathbb{F}_2$ be its m-th bit (LSB considered 0-th bit). Then, the consecutive k tuples of most significant v bits of the 32-bit integer output sequence is the rectangular part in:

$$k \text{ tuples} \left\{ \begin{array}{rcl} x_j[0] & : & \overbrace{b_{j,31}, b_{j,30}, ..., b_{j,31-v+1}}^{v \text{ bits}}, b_{j,31-v}, ..., b_{j,0} \\ x_j[1] & : & b_{j,63}, b_{j,62}, ..., b_{j,63-v+1}, b_{j,63-v}, ..., b_{j,32} \\ & & \vdots \\ x_j[3] & : & b_{j,127}, b_{j,126}, ..., b_{j,127-v+1}, b_{j,127-v}, ..., b_{j,96} \\ & & \vdots \\ x_{j+\lceil \frac{k}{4} \rceil}[k \mod 4] & : & ... \end{array} \right.$$

The corresponding part in the 128-bit sequence is marked by the parentheses in the following:

$$\left\lceil \frac{k}{4} \right\rceil \text{ tuples} \left\{ \begin{array}{rcl} x_j & : & \overbrace{b_{j,127}, ...,}^{v \text{ bits}} ..., \overbrace{b_{j,95}, ...,}^{v \text{ bits}} ..., \overbrace{b_{j,63}, ...,}^{v \text{ bits}} ..., \overbrace{b_{j,31}, ...,}^{v \text{ bits}} ..., b_{j,0} \\ \vdots & & \\ x_{j+\lceil \frac{k}{4} \rceil - 1} & : & b_{j,127}, ..., b_{j,95}, ..., \underbrace{\overbrace{b_{j,63}, ...,}^{v \text{ bits}} ..., \overbrace{b_{j,31}, ...,}^{v \text{ bits}} ..., b_{j,0}}_{(k \mod 4) \text{ 32-bit words}} \end{array} \right.$$

If we remove the last $v \times (k \mod 4)$ bits, then the rest $\frac{k}{4}4v$ bits among the 32-bit integers is identical to the same number of bits in the 128-bit integer sequence

$$\left\lceil \frac{k}{4} \right\rceil \text{ tuples} \left\{ \begin{array}{rcl} x_j & : & \overbrace{b_{j,31}, ...,}^{v \text{ bits}} \overbrace{b_{j,63}, ...,}^{v \text{ bits}} \overbrace{b_{j,95}, ...,}^{v \text{ bits}} \overbrace{b_{j,127}, ...,}^{v \text{ bits}} \\ & & \vdots \end{array} \right. \qquad (4)$$

We may apply the lattice method to this $4v$-bit sequence, and let k' be the dimension of the equidistribution. Then, we have an estimation of $k(v)$ as 32-bit integer sequence by

$$4k' \leq k(v) < 4(k' + 1).$$

To obtain the exact value of $k(v)$, we introduce another norm, named *weighted norm*, on F^{4v}. We consider a $4v$-bit integer sequence as in (4), and define the norm of weight type u ($u = 0, 1, 2, 3$) on F^{4v} as follows.

$$||(x_1, ..., x_{4v})||_u := \max\{\max_{i=1}^{(4-u)v}\{|x_i|\}, \max_{i=4v-uv+1}^{4v}\{2|x_i|\}\}$$

If $u = 0$, then this is the supremum norm treated already. It is easy to check that this gives an ultra norm for any u.

Theorem 3.8. *Let u to an integer with $0 \leq u \leq 3$, and let F^v equipped with the weighted norm of type u. Then, the covering radius of L with respect to this norm is $\leq 2^{-r-1}$, if and only if*

$$o(4r + u, 0) : V_p \to \mathbb{F}_2^{4r+u}$$

is surjective.

Proof. The surjectivity is equivalent to that there are enough points in L so that its $(4r+u)v$ bits corresponding to the weight-type u assume every possible bit pattern. This implies that the covering radius of L is at most 2^{-r-1}.

The converse follows in the same way. □

As explained above, by applying the usual (non-weighted) lattice method to $4v$-bit sequence, we obtain a closest lower bound $4k' \leq k(v) < 4(k'+1)$. So, using the above theorem for $u = 1, 2, 3$, we can check whether $o(4k'+u, 0)$ is surjective or not, to obtain the maximum k_0 such that $o(k_0, 0)$ is surjective.

A slightly modified method gives the maximum k_i such that $o(k_i, i)$ is surjective (for each $i = 1, 2, 3$). Now, $k(v)$ is obtained as the minimum of k_i, $i = 0, 1, 2, 3$. Thus, the algorithm to compute $k(v)$ is shown in Figure 3.

```
Input: $v$
Output: $k$
Loop $i$ = 0, 1, 2, 3
    Loop weight-type $u$ = 0, 1, 2, 3
        Compute the norm of the shortest basis of $4v$-bit integers
        with respect to the weight-type $u$. Let $2^{-r-1}$ be the norm
        of the shortest basis. Put $k_{i,u} := 4r + u$.
    End Loop
    Let $k_i$ be the maximum of $k_{i,u}$ $(u = 0, 1, 2, 3)$.
End Loop
Output the minimum value of $k_i$ $(i = 0, 1, 2, 3)$ as $k$.
```

Figure 3: algorithm of weighted norm method

We use Lenstra's reduction method to obtain a shortest basis from a generating set. Since the lattice L is independent of the weight type u, in this algorithm, after obtaining a shortest basis with weight type 0, we may apply Lenstra's algorithm to this shortest basis with respect to the weight-type 1. This is much faster than starting from the generating set in the definition of L.

A similar algorithm is applicable when SFMT19937 is considered as a 64-bit integer sequence generator.

4 Comparison of speed

We compared two algorithms: MT19937 and SFMT19937, with implementations using and without using SIMD instructions.

We measured the speeds for four different CPUs: Pentium M 1.4GHz, Pentium IV 3GHz, AMD Athlon 64 3800+, and PowerPC G4 1.33GHz. In returning the random values, we used two different methods. One is sequential generation, where one 32-bit random integer is returned for one call. The other is block generation, where an array of random integers is generated for one call (cf. [15]). For detail, see §2.4 above.

We measured the consumed CPU time in second, for 10^8 generations of 32-bit integers. More precisely, in case of the block generation, we generate 10^5 of 32-bit random integers by one call, and this is iterated for 10^3 times. For sequential generation, the same 10^8 32-bit integers are generated, one per call. We used the inline declaration `inline` to avoid the function call, and unsigned 32-bit, 64-bit integer types `uint32_t`, `uint64_t` defined in INTERNATIONAL STANDARD ISO/IEC 9899 : 1999(E) Programming Language-C, Second Edition (which we shall refer to as C99 in the rest of this thesis). Implementations without SIMD are written in C99, whereas those with SIMD use some standard SIMD extension of C99 supported by the compilers icl (Intel C compiler) and gcc.

Table 5 summarises the speed comparisons. The first four lines list the CPU time (in seconds) needed to generate 10^8 32-bit integers, for a Pentium-M CPU with the Intel C/C++ compiler. The first line lists the seconds for the block-generation scheme. The second line shows the ratio of CPU time to that of SFMT(SIMD). Thus, SFMT coded in SIMD is 2.10 times faster than MT coded in SIMD, and 3.77 times faster than MT without SIMD. The third line lists the seconds for the sequential generation scheme. The fourth line lists the ratio, with the basis taken at SFMT(SIMD) block-generation (not sequential). Thus, the block-generation of SFMT(SIMD) is 2.00 times faster than the sequential-generation of SFMT(SIMD).

Roughly speaking, in the block generation, SFMT(SIMD) is twice as fast as MT(SIMD), and four times faster than MT without using SIMD. Even in the sequential generation case, SFMT(SIMD) is still considerably faster than MT(SIMD).

Table 5: The CPU time (sec.) for 10^8 generations.

CPU/compiler	return	MT	MT(SIMD)	SFMT	SFMT(SIMD)
Pentium-M	block	1.122	0.627	0.689	0.298
1.4GHz	(ratio)	3.77	2.10	2.31	1.00
Intel C/C++	seq	1.511	1.221	1.017	0.597
ver. 9.0	(ratio)	5.07	4.10	3.41	2.00
Pentium IV	block	0.633	0.391	0.412	0.217
3GHz	(ratio)	2.92	1.80	1.90	1.00
Intel C/C++	seq	1.014	0.757	0.736	0.412
ver. 9.0	(ratio)	4.67	3.49	3.39	1.90
Athlon 64 3800+	block	0.686	0.376	0.318	0.156
2.4GHz	(ratio)	4.40	2.41	2.04	1.00
gcc	seq	0.756	0.607	0.552	0.428
ver. 4.0.2	(ratio)	4.85	3.89	3.54	2.74
PowerPC G4	block	1.089	0.490	0.914	0.235
1.33GHz	(ratio)	4.63	2.09	3.89	1.00
gcc	seq	1.794	1.358	1.645	0.701
ver. 4.0.0	(ratio)	7.63	5.78	7.00	2.98

This table shows the CPU time measured in second for 10^8 generations of 32-bit integers, for four different CPUs and two different return-value methods. The ratio to the SFMT coded in SIMD is listed, too.

Table 6 lists the CPU time for generating 10^8 32-bit integers, for four PRNGs from the GNU Scientific Library and two recent generators. They are re-coded with inline specification. Generators examined were: a multiple recursive gener-

Table 6: The CPU time (sec.) for 10^8 generations by six other PRNGs.

CPU	return	mrg	rand48	rand	random256g2	well	xor3
Pentium M	block	3.277	1.417	0.453	0.230	1.970	0.296
	seq	3.255	1.417	0.527	0.610	2.266	1.018
Pentium IV	block	2.295	1.285	0.416	0.121	0.919	0.328
	seq	2.395	1.304	0.413	0.392	1.033	0.702
Athlon	block	1.781	0.770	0.249	0.208	0.753	0.294
	seq	1.798	0.591	0.250	0.277	0.874	0.496
PowerPC	block	2.558	1.141	0.411	0.653	1.792	0.618
	seq	2.508	1.132	0.378	1.072	1.762	1.153

This table shows the CPU time (sec.) for 10^8 generations of 32-bit integers, by six other PRNGs.

Table 7: Dimension defects $d(v)$ of MT19937 and SFMT19937.

v	MT	SFMT	v	MT	SFMT	v	MT	SFMT	v	MT	SFMT
$d(1)$	0	0	$d(9)$	346	1	$d(17)$	549	543	$d(25)$	174	173
$d(2)$	0	*2	$d(10)$	124	0	$d(18)$	484	478	$d(26)$	143	142
$d(3)$	405	1	$d(11)$	564	0	$d(19)$	426	425	$d(27)$	115	114
$d(4)$	0	*2	$d(12)$	415	117	$d(20)$	373	372	$d(28)$	89	88
$d(5)$	249	2	$d(13)$	287	285	$d(21)$	326	325	$d(29)$	64	63
$d(6)$	207	0	$d(14)$	178	176	$d(22)$	283	282	$d(30)$	41	40
$d(7)$	355	1	$d(15)$	83	*85	$d(23)$	243	242	$d(31)$	20	19
$d(8)$	0	*1	$d(16)$	0	*2	$d(24)$	207	206	$d(32)$	0	*1

*This table shows Dimension defects $d(v)$ of MT19937 and SFMT19937 as a 32-bit integer generator. The mark * means that MT has a smaller defect than SFMT at that accuracy.*

ator `mrg` [16], linear congruential generators `rand48` and `rand`, a lagged fibonacci generator `random256g2`, a WELL generator `well` (WELL19937c in [28]), and a XORSHIFT generator `xor3` [27] [20]. The table shows that SFMT(SIMD) is faster than these PRNGs, except for the outdated linear congruential generator `rand`, the lagged-fibonacci generator `random256g2` (which is known to have poor randomness, cf. [23]), and `xor3` with a Pentium-M.

5 Dimension of equidistribution

Table 7 lists the dimension defects $d(v)$ of SFMT19937 (as a 32-bit integer generator) and of MT19937, for $v = 1, 2, \ldots, 32$. SFMT has smaller values of the defect $d(v)$ at 26 values of v. The converse holds for 6 values of v, but the difference is small. The total dimension defect Δ of SFMT19937 as a 32-bit integer generator is 4188, which is smaller than the total dimension defect 6750 of MT19937.

We also computed the dimension defects of SFMT19937 as a 64-bit (128-bit) integer generator, and the total dimension defect Δ is 14089 (28676, respectively). In some applications, the distribution of LSBs is important. To check them, we inverted the order of the bits (i.e. the i-th bit is exchanged with the $(w - i)$-th bit) in each integer, and computed the total dimension defect. It is 10328 (21337, 34577, respectively) as a 32-bit (64-bit, 128-bit, respectively) integer generator. Throughout the experiments, $d(v)$ is very small for $v \leq 10$. We consider that these values are satisfactorily small, since they are comparable with MT for which no statistical deviation related to the dimension defect has been reported, as far as we know.

Table 12 in Appendix shows the $k(v)$ and $d(v)$ (in parentheses) for all MEXPs of SFMT as 32-bit integer generators, and Table 13 and 14 show those as 64-bit integer generators.

6 Recovery from 0-excess states

For an LFSR with a sparse feedback function g, we observe the following phenomenon: if the bits in the state space contain too many 0's and few 1's (called a 0-excess state), then this tendency continues for many steps, since only a small part is changed in the state array at one step, and the change is not well-reflected to the next step because of the sparseness.

We measure the recovery time from 0-excess states, by the method introduced in [28], as follows.

1. Choose an initial state with only one bit being 1.
2. Generate k pseudorandom numbers, and discard them.
3. Compute the ratio of 1's among the next 32000 bits of outputs (i.e., in the next 1000 pseudorandom 32-bit integers).
4. Let γ_k be the average of the ratio over all such initial states.

We draw graphs of these ratio γ_k $(1 \leq k \leq 20000)$ in Figure 4 for the following generators: (1) WELL19937c, (2) PMT19937 (3) SFMT19937, and (4) MT19937.

Because of its dense feedback, WELL19937c shows the fastest recovery among the compared generators. SFMT is better than MT, since its recursion refers to two most recently computed words (`W[N-1]` and `W[N-2]`) that acquire new 1s, while MT refers only to the words generated long before (`W[M]` and `W[0]`). PMT19937 shows faster recovery than SFMT19937, since PMT19937 has two feedback loops. The speed of recovery from 0-excess states is a trade-off with the speed of generation. Such 0-excess states will not happen practically, since the probability that 19937 random bits have less than 19937×0.4 of 1's is about 5.7×10^{-177}. The only plausible case would be that a poor initialization scheme gives a 0-excess initial state (or gives two initial states whose Hamming distance is too small). In a typical simulation, the number of initializations is far smaller than the number of generations, therefore we may spend more CPU time in the initialization than the generation. Under the assumption that a good initialization scheme is provided, the slower recovery of SFMT compared to WELL would perhaps not be a great issue.

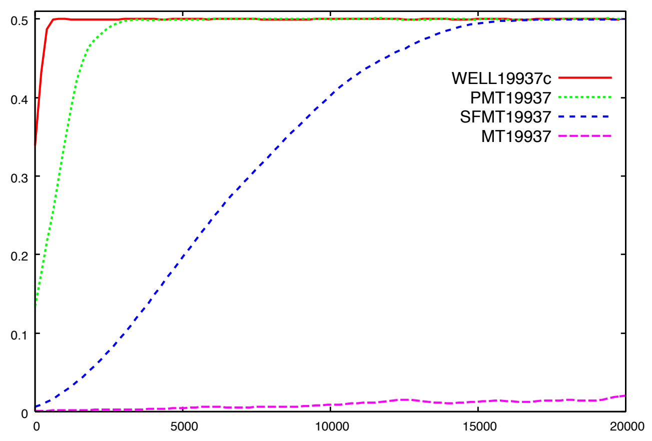

Figure 4: γ_k $(k = 0, \ldots, 20000)$: Starting from extreme 0-excess states.

γ_k $(k = 0, \ldots, 20000)$: Starting from extreme 0-excess states, discard the first k outputs and then measure the ratio γ_k of 1's in the next 1000 outputs. In the order of the recovery speed: (1) WELL19937c, (2) PMT19937, (3) SFMT19937, and (3) MT19937.

7 Portability

Using CPU dependent features causes a portability problem. We prepare (1) a standard C code of SFMT, which uses functions specified in C99 only, (2) an optimized C code for Intel Pentium SSE2, and (3) an optimized C code for PowerPC AltiVec. The optimized codes require icl (Intel C Compiler) or gcc compiler with suitable options. Here we mention again that SFMT implemented in standard C code is faster than MT.

There is a problem of the endian when 128-bit integers are converted into 32-bit integers. When a 128-bit integer is stored as an array of 32-bit integers with length 4, in a little endian system such as Pentium, the 32 LSBs of the 128-bit integer come first. On the other hand, in a big endian system such as PowerPC, the 32 MSBs come first. The explanation above is based on the former. To assure the exactly same outputs for both endian systems as 32-bit integer generators, in the SIMD implementation for PowerPC, the recursion (2) is considered as a recursion on quadruples of 32-bit integers, rather than 128-bit integers, so that the content of the state array coincides both for little and big endian systems, as an array of 32-bit integers (not as 128-bit integers). Then, shift operations on 128-bit integers in PowerPC differs from those of Pentium. Fortunately, PowerPC supports arbitrary permutations of 16 blocks of 8-bit integers in a 128-bit register, which emulates the Pentium's shift by a multiple of 8.

8 Conclusions of Chapter 1

We proposed the SFMT pseudorandom number generator, which is a very fast generator with satisfactorily high-dimensional equidistribution property.

8.1 Trade-off between speed and quality

It is difficult to measure the generation speed of a PRNG in a fair way, since it depends heavily on the circumstances. The WELL [28] generators have the best possible dimensions of equidistribution (i.e. $\Delta = 0$) for various periods ($2^{1024}-1$ to $2^{19937}-1$). If we use the function call to the PRNG for each generation, then a large part of the CPU time is consumed for handling the function call, and in the experiments in [28] or [27], WELL is not much slower than MT. On the other hand, if we avoid the function call, WELL is slower than MT for some CPUs, as seen in Table 5.

Since $\Delta = 0$, WELL has a better quality than MT or SFMT in a theoretical sense. However, one may argue whether this difference is observable or not. In the case of an $\mathbb{F}_2$-linear generator, the dimension of equidistribution $k(v)$ of v-bit accuracy means that there is no constant linear relation among the kv bits, but there exists a linear relation among the $(k+1)v$ bits, where kv bits ($(k+1)v$ bits) are taken from all the consecutive k integers ($k+1$ integers, respectively) by extracting the v MSBs from each. However, the existence of a linear relation does not necessarily mean the existence of some observable bias. According to [22], it requires 10^{28} samples to detect an $\mathbb{F}_2$-linear relation with 15 (or more) terms among 521 bits, by weight distribution test. If the number of bits is increased, the necessary sample size is increased rapidly. Thus, it seems that $k(v)$ of SFMT19937 is sufficiently large, far beyond the level of the observable bias. On the other hand, the speed of the generator is observable. Thus, SFMT focuses more on the speed, for applications that require fast generations. (Note: a referee of [29] pointed out that statistical tests based on the rank of $\mathbb{F}_2$-matrix is sensitive to the linear relations [19], so the above observation is not necessarily true.)

Chapter 2

A fast stream cipher, CryptMT

In this chapter, we pursue a fast stream cipher in software. We assume that the machine has plenty of memory, and a fast integer multiplication instruction.

9 Stream Cipher

Let B be the set of symbols. Throughout this chapter, we assume B to be the set of one byte integers, which is identified with $\mathbb{F}_2^8$, where $\mathbb{F}_2 = \{0, 1\}$ is the two-element field. We consider a stream cipher based on a key-stream generator over B. A generator receives a key k in the set of possible keys K, then generates a sequence of elements

$$b_0(k), b_1(k), \ldots, b_n(k), \ldots, \in B.$$

A plain text (a sequence of elements of B) is encrypted by taking bitwise exor with the sequence $(b_n(k))$, and then decrypted by the same method.

9.1 Combined Generator

Such a sequence is typically generated by a finite state automaton.

Definition 9.1. *A finite state automaton A without input is a quadruple $A = (S, f, O, o)$, where S is a finite set (the set of states), $f : S \to S$ is a function (the state transition function), O is a set (the set of the output symbols), and $o : S \to O$ is the output function.*

For a given initial state s_0, A changes the state by the recursion $s_n := f(s_{n-1})$ $(n = 1, 2, 3, \ldots)$ and generates the sequence

$$o(s_0), o(s_1), o(s_2), \ldots \in O.$$

For a stream cipher, we prepare an *initializing* function $\mathrm{init} : K \to S$, and take $O := B$. By setting $s_0 := \mathrm{init}(k)$, the automaton A generates a sequence of elements in B. Its period is bounded by $\#(S)$.

To obtain a secure generator, larger $\#(S)$ and complicated f and o are desirable. However, if f is complicated, then the analysis of the sequence (such as computing the period and the distribution) often becomes difficult. A typical choice is to choose an $\mathbb{F}_2$-linear transition function. We take $S := \mathbb{F}_2^d$ and choose a linear transition function f. Then, the period can be computed by the linear algebra and polynomial calculus. In particular, the following linear feedback shift register generators (LFSRs) are widely used: $S := (\mathbb{F}_2^w)^n$ where w is the word size of the machine (e.g. $w = 32$ for 32-bit machines), and the transition is

$$f(x_1, x_2, \ldots, x_{n-1}, x_n) := (x_2, x_3, \ldots, x_n, g(x_1, \ldots, x_n)). \tag{5}$$

Here $g : (\mathbb{F}_2^w)^n \to \mathbb{F}_2^w$ is a linear function called the feedback function. This state transition is equivalent to the recursion

$$x_{i+n} := g(x_i, x_{i+1}, \ldots, x_{i+n-1}) \quad (i = 0, 1, 2, \ldots).$$

The output of LFSR is given by

$$o : S \to \mathbb{F}_2^w, \quad (x_1, \ldots, x_n) \mapsto x_1,$$

which is not secure as it is. A software implementation technique using a cyclic array ([15, P.28 Algorithm A]) reduces the computation of f to that of g and an index change. Consequently, the computation time is independent of the size of n, which allows a fast generator with huge state space. This type of generator is common for the pseudorandom number generation in Monte Carlo method (PRNG for MC), such as Mersenne Twister (MT19937) [21], whose period is $2^{19937} - 1$.

As a stream cipher, any linear recurring sequence is vulnerable, so we need to introduce some non-linearity. A conventional method is to choose a "highly non-linear" $o : S \to O$. In this context, o is called *a filter*.

One of the estimators of the non-linear property of a function is the *algebraic degree*.

Definition 9.2. *Let $h(c_1, c_2, \ldots, c_n)$ be a boolean function, i.e.,*

$$h : \mathbb{F}_2^n \to \mathbb{F}_2.$$

Then, the function h can be represented as a polynomial function of n variables $c_1, c_2, \ldots, c_n$ with coefficients in $\mathbb{F}_2$, namely as a function

$$h = \sum_{T \subset \{1,2,\ldots,n\}} a_T c_T$$

holds, where $a_T \in \mathbb{F}_2$ and $c_T := \prod_{t \in T} c_t$. This representation is unique, and called the algebraic normal form *of h. Its degree is called the algebraic degree of h.*

Let $h_{i,n}(s_0)$ denote the i-th bit of the n-th output $b_n(s_0)$ of the generator for the initial state s_0. This is a boolean function, when we consider $s_0 \in S = \mathbb{F}_2^d$ as d variables of bit. Thus, an adversary can obtain s_0 by solving the simultaneous equations $h_{i,n}(s_0) = o_{i,n}$ for unknown s_0 for various i and n, where $o_{i,n}$ are the outputs of the generator observed by the adversary. This is the *algebraic attack* (see for example [6], [5]).

A problem of a linear generator with filter is the following. Since any s_n is a linear function of the bits in $s_0 = \text{init}(k)$, the algebraic degree of $h_{i,n}(s_0)$ is bounded from the above by the algebraic degree of the i-th bit of the filter function o, namely that of the function

$$o_i : S \xrightarrow{o} \mathbb{F}_2^8 \xrightarrow{i\text{th}} \mathbb{F}_2.$$

To attain the high-speed generation, o_i cannot access so many bits in S, and its algebraic degree is bounded by the number of accessed bits. This decreases the merits of the large state space. A *filter with memory*, which is just a finite state automaton with input, solves this conflict (see §10.3 for its effect on the algebraic degree).

Definition 9.3. *A finite state automaton A with input is a five-tuple $A = (S, I, f, O, o)$. The data S, O, o are same with Definition 9.1. The difference is*

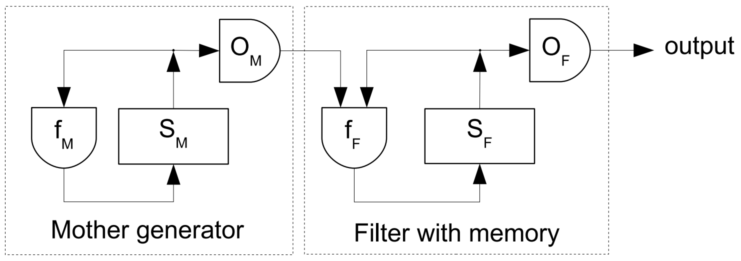

Figure 5: Combined generator.

that it has another component I (the set of input symbols), and that the state transition function is of the form $f : I \times S \to S$. For an initial state s_0 and an input sequence $i_0, i_1, \ldots \in I$, A changes the state by $s_n = f(i_{n-1}, s_{n-1})$ $(n = 1, 2, 3, \ldots)$.

Definition 9.4. *(A combined generator with filter with memory.)*

Let $A_M := (S_M, f_M, O_M, o_M)$ be an automaton without input (called the mother generator*, M for mother). Let $A_F := (S_F, I_F, f_F, O_F, o_F)$ be an automaton with input (called the* filter with memory*, F for the filter). We assume that $O_M = I_F$. Consider a pair of initial states $s_{M,0} \in S_M$ and $s_{F,0} \in S_F$. We generate a sequence of $O_M = I_F$ by A_M with initial state $s_{M,0}$, and pass it to A_F with initial state $s_{F,0}$, to obtain a sequence of O_F as the output sequence. This amounts to considering an automaton C without input, named the* combined generator*: the state space S_C of C is $S_M \times S_F$, the transition function is*

$$f_C : (s_M, s_F) \mapsto (f_M(s_M), f_F(o_M(s_M), s_F)),$$

and the output function is

$$o_C : (s_M, s_F) \mapsto o_F(s_F) \in O_F.$$

Figure 5 describes a combined generator.

Example 9.5. *The output function o_C in the above definition depends only on S_F, but we may consider a function depending both S_M and S_F.*

Such an example is famous SNOW stream cipher [8] [9]. The mother generator of SNOW2.0 is an LFSR with 512-bit state space, and its filter has 64-bit state space. Non-linearity is introduced by four copies of one same S-box of 8-bit size, based on arithmetic operations in 2^8-element filed $\mathbb{F}_{2^8}$.

SNOW has no rigorous assurance on the period and the distribution of the generated sequence. We shall introduce the notion of quasigroup filter, which allows to compute the period and distribution property.

10 Quasigroup Filter

Definition 10.1. *A function $f : X \times Y \to Z$ is said to be bi-bijective if $f(-, y) : X \to Z$, $x \mapsto f(x, y)$ is bijective for any fixed y, and so is $f(x, -) : Y \to Z$, $y \mapsto f(x, y)$ for any fixed x. If $X = Y = Z$, this coincides with the notion of a quasigroup.*

A quasigroup filter is an automaton in Definition 9.3 where the state transition function $f : I \times S \to S$ is bi-bijective.

Example 10.2. *(Multiplicative filter).*

Let $I = S$ be the set of odd integers in the ring $\mathbb{Z}/2^{32}$ of integers modulo 2^{32}. Let $f : I \times S \to S$ be the integer multiplication modulo 2^{32}. This is a quasigroup (actually the multiplicative group of the ring $\mathbb{Z}/2^{32}$).

We choose $o_F : S \to O_F = B$ as the function taking the 8 MSBs from the 32-bit integer.

Example 10.3. *If we correspond a 32-bit integer x to a 33-bit odd integer $2x+1$ modulo 2^{33}, then the multiplication formula*

$$(2x+1) \times (2y+1) = 2(2xy + x + y) + 1$$

gives a quasigroup structure

$$\tilde{\times} : (x, y) \mapsto x \tilde{\times} y := 2xy + x + y \mod 2^{32}$$

on the set of 32-bit integers. We can consider the corresponding multiplicative filter with $I = S$ being the set of 32-bit integers.

Modern CPUs often have a fast integer multiplication for 32-bit integers. We shall discuss mathematical property of such filters in §10.3.

Example 10.4. *(CryptMT1: MT with multiplicative filter)*

We choose an LFSR described in (5) as the mother generator A_M, with $O_M = \mathbb{F}_2^w$. We can choose its parameters so that the period is a large Mersenne prime $Q = 2^p - 1$ (e.g. $p = 19937$ as in the case of MT19937 [21]). By identifying O_M as the set of w-bit integers, we can use the multiplicative filter A_F described in Example 10.3. We call this generator as MT19937 with multiplicative filter. *The output function $o_F : S_F \to O_F = \mathbb{F}_2^8$ is extracting 8 MSBs. This generator is called CryptMT Version 1 (CryptMT1) [24].*

10.1 k-dimensional Distribution

Let k be an integer, and let A be an automaton without input as in Definition 9.1. We define its *k-tuple output function* $o^{(k)}$ by

$$o^{(k)} : S \to O^k \quad s \mapsto (o(s), o(f(s)), o(f^2(s)), \ldots, o(f^{k-1}(s))) \tag{6}$$

(i.e. $o^{(k)}$ maps the state to the next k outputs). Consider the multi-set of the possible output k-tuples for all states:

$$O^{(k)} := \{o^k(s) \mid s \in S\}.$$

This is the image of S by $o^{(k)}$ counted with multiplicities.

Definition 10.5. *The output of the automaton A is said to be k-dimensionally equidistributed if the multiplicity of each element in $O^{(k)}$ is same.*

This type of criteria is commonly used for PRNG for MC: MT19937 as a 32-bit integer generator has this property with $k = 623$. This criterion is equivalent to the uniformness of the function $o^{(k)}$ defined below.

Definition 10.6. *A mapping $g : X \to Y$ is* uniform *if the cardinality of $g^{-1}(y)$ is independent of $y \in Y$. A bijection is uniform, and the composition of uniform mappings is uniform.*

A filter with memory is uniform *if its output function is uniform.*

Example 10.4 is uniform. The next proposition shows that a uniform quasigroup filter increases the dimension of equidistribution by 1.

Proposition 10.7. *We keep the set-up of Definition 9.4. Assume that A_F is a uniform quasigroup filter. Suppose that the output of A_M is k-dimensionally equidistributed. Then, the combined generator C is $(k+1)$-dimensionally equidistributed.*

Proof. Consider the k-tuple output function of the mother generator $o_M^{(k)} : S_M \to O_M^k$ as in (6). Then, the k-dimensional equidistribution property is equivalent to the uniformness of $o_M^{(k)}$. The $(k+1)$-tuple output function $o_C^{(k+1)}$ of the combined generator C is the composite

$$o_C^{(k+1)} : S_M \times S_F \xrightarrow{o_M^{(k)} \times \mathrm{Id}_{S_F}} O_M^k \times S_F \xrightarrow{\mu} S_F^{k+1} \xrightarrow{o_F^{k+1}} O_F^{k+1},$$

where the second map μ is given by

$$\mu : ((x_k, x_{k-1}, \ldots, x_1), y_1) \mapsto (y_{k+1}, y_k, \ldots, y_1)$$

where y_i's are inductively defined by $y_{i+1} := f_F(x_i, y_i) \quad (i = 1, 2, \ldots, k)$. The last map o_F^{k+1} is the direct product of $k+1$ copies of o_F. The quasigroup property of f_F implies the bijectivity of μ. The last map is uniform. Since the composition of uniform mappings is uniform, we obtain the proof. □

Corollary 10.8. *CryptMT1 explained in Example 10.4 is 624-dimensionally equidistributed.*

We mean by a *simple distinguishing attack of order N* to choose a real function F with N variables and to detect the deviation of the distribution of the values of F applied to the consecutive N-outputs. If N does not exceed the dimension of the equidistribution, then one can observe no deviation from the true randomness, under the assumption of uniform choice of the initial state.

By this reason, it seems very difficult to apply a correlation attack or a distinguishing attack to such generators. For example, to observe some deviation of MT19937 with multiplicative filter in Example 10.4, one needs to observe the correlation of outputs with the lag more than 624. Because of the high nonlinearity of the multiplicative filter discussed below, we guess that this would be infeasible.

10.2 A Theorem on the Period

Theorem 10.9. *Consider a combined generator C as in Definition 9.4. Let $s_{M,0}$ be the initial state of the mother generator A_M, and assume that its state transition is purely periodic with period $P = Qq$ for a prime Q and an integer q. Let $S^o \subset S_M$ be the orbit of the state transition. Let k be an integer. Assume that the k-tuple output function of the mother generator $o_M^{(k)} : S^o \to O_M^k$*

as defined in (6) is surjective when restricted to S^o. Suppose that A_F is a quasigroup filter as in Definition 10.7.

Let r be the ratio of the occupation of the maximum inverse image of one element by $o_F : S_F \to O_F$ in S_F, namely

$$r = \max_{b \in O_F} \{\#(o_F^{-1}(b))\}/\#(S_F).$$

If

$$r^{-(k+1)} > q \cdot (\#(S_F))^2,$$

then the period of the output sequence of C is a nonzero multiple of Q.

Proof. We may replace S_M with the orbit starting from s_0. Then, replace S_M with its quotient set where two states are identified if the output sequences from them are identical. Thus, we may assume $\#(S_M) = P$, where P is the period of the output sequence of A_M. In this proof, we do not consider multi-sets. Consider the k-tuple output function $o_C^{(k+1)}$ as in the proof of Proposition 10.7:

$$o_C^{(k+1)} : S_M \times S_F \xrightarrow{o_M^{(k)} \times \mathrm{Id}_{S_F}} O_M^k \times S_F \xrightarrow{\mu} S_F^{k+1} \xrightarrow{o_F^{k+1}} O_F^{k+1}.$$

Since $o_M^{(k)}$ is surjective and μ is bijective (by the quasigroup property), the image $I \subset O_M^{n+1}$ of $S_M \times \{y_0\}$ by $\mu \circ (o_M^{(k)} \times \mathrm{Id}_Y)$ has the cardinality $\#(O_M)^k$. By the assumption of the pure periodicity of x_i and the bijectivity of f_F, the output sequence $o_F(y_i)$ $(i = 0, 1, 2, \ldots)$ is purely periodic. Let p be its period. Then, $o_F^{k+1}(I) \subset O_F^{k+1}$ can have at most p elements. o_F is uniform, so we can regard it as n-to-one correspondence, then o_F^{k+1} is n^{k+1}-to-one correspondence. And then $\frac{\#(I)}{\#(o_F^{k+1}(I)} \le n^{k+1}$ because $I \subset S_F^{k+1}$. Therefore $\#(I) \le pn^{k+1}$. The definition of r means $n = r\#(S_F)$, then

$$\#(I) \le p(r\#(S_F))^{k+1}.$$

Since $\#(O_M)^k = \#(I)$ and $\#(O_M) = \#(S_F)$, we have an inequality

$$r^{-(k+1)} \le p\#(S_F).$$

The period P' of the state transition of C is a multiple of $P = Qq$. Since the state size of C is $P \times \#(S_F)$, $P' = Qm$ holds for some $m \le q\#(S_F)$. Consequently, p is a divisor of Qm. If p is not a multiple of Q, then p divides m, and then $p \le q\#(S_F)$. Thus we have

$$r^{-(k+1)} \le q\#(S_F)^2,$$

contradicting to the assumption. □

Example 10.10. *For MT19937 with multiplicative filter, this theorem shows that any bit in the output sequence has a period being a multiple of the prime $2^{19937} - 1$, as follows.*

We have $Q = 2^{19937} - 1$ and $q = 1$. If $o_F : S_F \to O_F = \mathbb{F}_2^m$ is extracting some m bits from the 32-bit integers, then $r = 2^{w-m}/2^w = 2^{-m}$. The inequality condition in the theorem is now

$$2^{m(k+1)} > 2^{2w},$$

and hence if this holds, then the m-bit output sequence has a period which is a multiple of Q.

In the case of MT19937 and the multiplicative filter, since $k = 623$ and $w = 32$, the above inequality holds for any $m \geq 1$, hence any bit of the output has a period at least $2^{19937} - 1$.

10.3 A Proposition on the Algebraic Degree of Integer Products

Definition 10.11. *Let us define a boolean function $m_{s,N}$ of $(s-1)N$ variables, as follows. Consider N of s-bit integer variables $x_1, \ldots, x_N$. Let*

$$c_{s-1,i}c_{s-2,i} \cdots c_{0,i}$$

be the 2-adic representation of x_i, hence $c_{j,i} = 0, 1$. We fix $c_{0,i} = 1$ for all $i = 1, \ldots, N$, i.e. assuming x_i odd. The boolean function $m_{s,N}$ has variables $c_{j,i}$ $(j = 1, 2, \ldots, s-1, i = 1, 2, \ldots, N)$, and its value is defined as the s-th digit (from the LSB) of the 2-adic expansion of the product $x_1 x_2 \cdots x_N$ as an integer.

Proposition 10.12. *Assume that $N, s \geq 2$. The algebraic degree of $m_{s,N}$ is bounded from below by*

$$\min\{2^{s-2}, 2^{\lfloor \log_2 N \rfloor}\}.$$

Let $h(c_1, c_2, \ldots, c_n)$ be a boolean function as in Definition 9.2, and $h = \sum_{T \subset \{1,2,\ldots,n\}} a_T c_T$ be its algebraic normal form.

The following lemma is well known.

Lemma 10.13. *It holds that $a_T = \sum_{U \subset T} h(U)$, where $h(U) := h(c_1, \ldots, c_n)$ with $c_i = 0, 1$ according to $i \notin U$, $\in U$, respectively.*

Proof of Proposition. For $s = 2$, the claim is easy to check. We assume $s \geq 3$.

Case 1. $s - 2 \leq \log_2 N$. In this case, it suffices to prove that the algebraic degree is at least 2^{s-2}. Take a subset T of size 2^{s-2} from $\{1, 2, \ldots, N\}$, say $T = \{1, 2, \ldots, 2^{s-2}\}$. Then, we choose $c_{1,1}, c_{1,2}, \ldots, c_{1,2^{s-2}}$ as the $\#T$ variables "activated" in Lemma 10.13, and consequently, the coefficient of $c_{1,1}c_{1,2} \cdots c_{1,2^{s-2}}$ in the algebraic normal form of $m_{s,N}$ is given by the sum in $\mathbb{F}_2$:

$$a_T := \sum_{U \subset T} (s\text{-th bit of } x_1 \cdots x_n, \text{ where } c_{j,i} = 1 \text{ if and only if } j = 1 \text{ and } i \in U).$$

Note that $c_{0,i} = 1$. It suffices to prove $a_T = 1$. Now, each term in the right summation is the s-th bit of the integer $3^{\#U}$, so the right hand side equals to

$$\sum_{m=0}^{2^{s-2}} \left[\binom{2^{s-2}}{m} \times \text{ the } s\text{-th bit of } 3^m \right].$$

However, the well-known formula

$$(x+y)^{2^{s-2}} \equiv x^{2^{s-2}} + y^{2^{s-2}} \mod 2$$

implies that the binary coefficients are even except for the both end, so the summation is equal to the s-th bit of $3^{2^{s-2}}$.

A well-known lemma says that if $x \equiv 1 \mod 2^i$ and $x \not\equiv 1 \mod 2^{i+1}$ for $i \geq 2$, then $x^2 \equiv 1 \mod 2^{i+1}$ and $x^2 \not\equiv 1 \mod 2^{i+2}$. By applying this lemma inductively, we know that

$$3^{2^{s-2}} = (1+8)^{2^{s-3}} \equiv 1 \mod 2^s, \quad \not\equiv 1 \mod 2^{s+1}.$$

This means that s-th bit of $3^{2^{s-2}}$ is 1, and the proposition is proved.

Case 2. $s-2 > \lfloor \log_2(N) \rfloor$. In this case, we put $t := \lfloor \log_2(N) \rfloor + 2$, and hence $s > t$ and $2^{t-2} \leq N$. We apply the above arguments for $T = \{1, 2, \ldots, 2^{t-2}\}$, but this time instead of $c_{1,i}$, we activate

$$\{c_{s-t+2,i} \mid i \in T\}.$$

The same argument as above reduces the non-vanishing of the coefficient of the term $c_{s-t+2,1} \cdots c_{s-t+2,2^{t-2}}$ to the non-vanishing of

$$\sum_{m=0}^{2^{t-2}} \left[\binom{2^{t-2}}{m} \times \text{the } s\text{-th bit of } (1+2^{s-t+2})^m \right].$$

Again, only the both ends $m = 0$ and $m = 2^{t-2}$ can survive, and the above summation is the s-th bit of $(1+2^{s-t+2})^{t-2}$. Since $s - t + 2 \geq 2$, the lemma mentioned above implies that

$$(1+2^{s-t+2})^{2^{t-2}} \equiv 1 \bmod 2^s, \quad \not\equiv 1 \bmod 2^{s+1},$$

which implies that its s-th bit is 1. □

This proposition gives the algebraic degree of the multiplicative filter, with respect to the inputs $x_1, \ldots, x_N$.

This proposition implies that we should use MSBs of the multiplicative filter. On the other hand, using 8 MSBs among 32-bit integers as in Example 10.2 seems to have enough high algebraic degree. We check this using a toy model.

10.4 Simulation by Toy Models

Since the filter has a memory, it is not clear how to define the algebraic degree or non-linearity of the filter. Instead, if we consider all bits in the initial state as variables, then each bit of the outputs is a boolean function of these variables, and algebraic degree and non-linearity are defined.

However, it seems difficult to compute them explicitly for CryptMT3, because of the size. So we made a toy model and obtained experimental results. Its mother generator is a linear generator with 16-bit internal state, and generates a 16-bit integer sequence defined by

$$\mathbf{x}_{j+1} := (\mathbf{x}_j >> 1) \oplus ((\mathbf{x}_j \& 1) \cdot \mathbf{a}),$$

where $>> 1$ denotes the one-bit shift-right, $(\mathbf{x}_j \& 1)$ denotes the LSB of $\mathbf{x}_j$, $\mathbf{a} = 1010001001111000$ is a constant 16-bit integer, and $(\mathbf{x}_j \& 1) \cdot \mathbf{a}$ denotes the product of the scalar $(\mathbf{x}_j \& 1) \in \mathbb{F}_2$ and the vector $\mathbf{a}$.

Then it is filtered by

$$y_{j+1} = (\mathbf{x}_j | 1) \times y_j \mod 2^{16},$$

Table 8: Table of the algebraic degrees of output bits of a toy model.

y_1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
y_2	14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	0
y_3	15	15	14	13	12	11	10	9	8	6	4	3	2	1	1	0
y_4	15	16	15	14	13	12	11	10	9	7	5	4	2	1	1	0
y_5	16	16	15	15	14	13	12	11	10	7	5	4	2	1	1	0
y_6	16	16	15	15	15	14	13	11	10	9	7	4	2	1	1	0
y_7	16	15	16	16	15	15	14	13	12	9	7	4	2	1	1	0
y_8	15	15	15	16	16	15	15	14	13	10	8	4	2	1	1	0
y_9	16	15	16	15	15	16	15	15	13	10	8	4	2	1	1	0
y_{10}	15	16	16	16	16	16	15	15	14	12	8	4	2	1	1	0
y_{11}	15	16	16	15	15	15	16	15	15	12	8	4	2	1	1	0
y_{12}	15	16	16	16	16	15	16	16	15	13	8	4	2	1	1	0
y_{13}	16	15	15	15	15	15	16	15	16	13	8	4	2	1	1	0
y_{14}	15	15	16	15	15	16	16	15	16	15	8	4	2	1	1	0
y_{15}	15	16	16	16	15	16	16	16	15	14	8	4	2	1	1	0
y_{16}	16	15	16	15	15	15	15	15	16	14	8	4	2	1	1	0

Table 9: The non-linearity of the MSB of each output of a toy model.

output	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9
nonlinearity	0	32112	32204	32238	32201	32211	32208	32170	32235

where $(\mathbf{x}_j|1)$ denotes $\mathbf{x}_j$ with LSB set to 1. We put $y_0 = 1$, and compute the algebraic degree of each of the 16 bits in the outputs $y_1 \sim y_{16}$, each regarded as a polynomial function with 16 variables being the bits in $\mathbf{x}_0$. The result is listed in Table 8. The lower six bits of the table clearly show the pattern $0, 1, 1, 2, 4, 8$, which suggests that the lower bound 2^{s-2} for $s \geq 2$ given in Proposition 10.12 would be tight, when the iterations are many enough. On the other hand, eighth bit and higher are "saturated" to the upper bound 16, after 12 generations.

We expect that the algebraic degrees for CryptMT3 would behave even better, since its filter is modified. So, if we consider each bit of the internal state of CryptMT3 as a variable, then the algebraic degree of the bits in the outputs will be near to 19937, after some steps of generations.

Also, we computed the non-linearity of the MSB of each y_i $(i = 1, 2, \ldots, 8)$ of this toy model. The result is listed in Table 9, and each value is near to 2^{16-1}. This suggests that there would be no good linear approximation of CryptMT3.

11 A Fast Initialization of a Large State Space

Consider LFSR in (5) as a mother generator. Its state space is an array of w-bit integers with size n. We need to give initial values to such a large array in the initialization. If one wants to encrypt a much shorter message than n, then this is not efficient. A possible solution is to use a PRNG with relatively small state space (called *the booter*) which can be quickly initialized, and use it to generate the initial array $x_0, x_1, \ldots, x_{n-1}$, and at the same time, its outputs are passed to the filter for key-stream generation. If the message length is smaller than n,

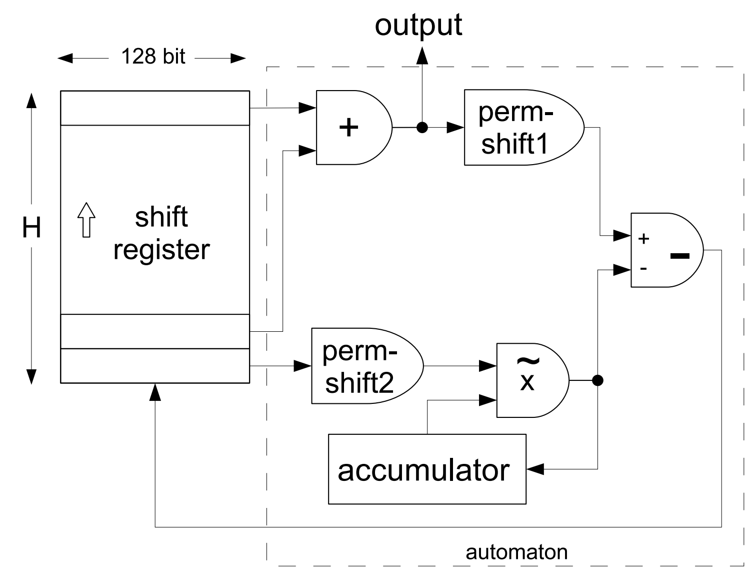

Figure 6: Booter of CryptMT3.

perm-shift1: $\mathbf{x} \mapsto (\mathbf{x}[2][1][0][3]) \oplus (\mathbf{x} >>_{32} 13)$.
perm-shift2: $\mathbf{x} \mapsto (\mathbf{x}[1][0][2][3]) \oplus (\mathbf{x} >>_{32} 11)$.
$\tilde{\times}$: multiplication of (a quadruple of) 33-bit odd integers.
$+, -$: addition, subtraction of four 32-bit integers modulo 2^{32}.

then the mother generator is never used: the outputs of the booter are used as the output of the mother generator. If the message length exceeds n, then the first n outputs of the booter are used as the outputs of the mother generator, and at the same time for filling up the state space of the mother generator. After the state space is filled up, the mother generator starts to work.

11.1 The key, IV, and the Booter

The design of the booter (see §11) goes independently of the key-stream generator. However, as the referees pointed out, we need to specify one to have a complete description of the generator. Thus, we here include the booter of CryptMT3 for self-containedness. The booter is described in Figure 6.

We choose an integer H later in §11.2. The state space of the booter is a shift register consisting of H 128-bit integers. We choose an initial state $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{H-1}$ and the initial value $\mathbf{a}_0$ of the accumulator (a 128-bit memory) as described in the next section. Then, the state transition is given by the recursion

$$\begin{aligned} \mathbf{a}_j &:= (\mathbf{a}_{j-1} \tilde{\times}_{32} \texttt{perm-shift2}(\mathbf{x}_{H+j-1})) \\ \mathbf{x}_{H+j} &:= \texttt{perm-shift1}(\mathbf{x}_j +_{32} \mathbf{x}_{H+j-2}) -_{32} \mathbf{a}_j, \end{aligned}$$

where

$$\begin{aligned} \texttt{perm-shift1}(\mathbf{x}) &:= (\mathbf{x}[2][1][0][3]) \oplus (\mathbf{x} >>_{32} 13) \\ \texttt{perm-shift2}(\mathbf{x}) &:= (\mathbf{x}[1][0][2][3]) \oplus (\mathbf{x} >>_{32} 11). \end{aligned}$$

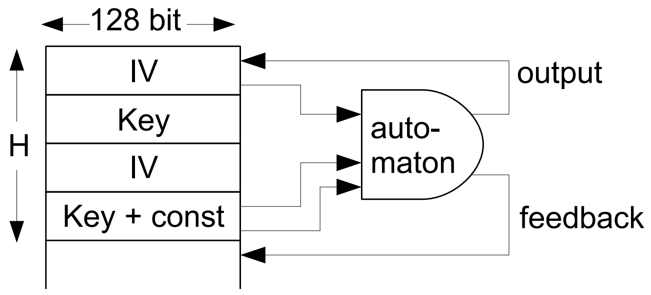

Figure 7: Beginning of Key and IV set-up.

The IV-array and Key-array are concatenated and copied to an array twice. Then, a constant is added to the bottom of the second copy of the key to break a possible symmetry. The automaton is described in Figure 6.

The notation $+_{32}$ ($-_{32}$) denotes the addition (subtraction, respectively) modulo 2^{32} for each of the four 32-bit integers in the 128-bit integers. The output of the j-th step is $\mathbf{x}_j +_{32} \mathbf{x}_{H+j-2}$.

As described in Figure 6, the booter consists of an automaton with three inputs and two outputs of 128-bit integers, together with a shift register. In the implementation, the shift register is taken in an array of 128-bit integers with the length $2H + 2 + n$, where $n = 156$ is the size of the state array of SFMT. This redundancy of the length is for the idling, as explained below.

11.2 Key and IV Set-up

We assume that both the IV and the Key are given as arrays of 128-bit integers. The size of each array is chosen by user, from 1 to 16. Thus, the Key-size is chosen from 128 bits to 2048 bits, as well as the IV-size. We claim the security level that is the same with the minimum of the Key-size and the IV-size.

We concatenate the IV and the Key to a single array, and then it is copied twice to an array, as described in Figure 7.

To break the symmetry, we add a constant 128-bit integer (846264, 979323, 265358, 314159) (denoting four 32-bit integers in a decimal notation, coming from π) to the bottom row of the second copy of the key (add means $+_{32}$). Now, the size H of the shift register in the booter is set to be $2 \times$ (IV-size + Key-size (in bits))/128, namely, the twice of the number of 128-bit integers contained in the IV and the Key. For example, if the IV-size and the Key-size are both 128 bits, then $H = 2 \times (1 + 1) = 4$. The automaton in the booter described in Figure 6 is equipped on this array, as shown in Figure 7. The accumulator of the booter-automaton is set to

$$\text{(the top row of the key array)} \mid (1, 1, 1, 1),$$

that is, the top row is copied to the accumulator and then the LSB of each of the 32-bit integers in the accumulator is set to 1.

At the first generation, the automaton reads three 128-bit integers from the array, and write the output 128-bit integer at the top of the array. The feedback

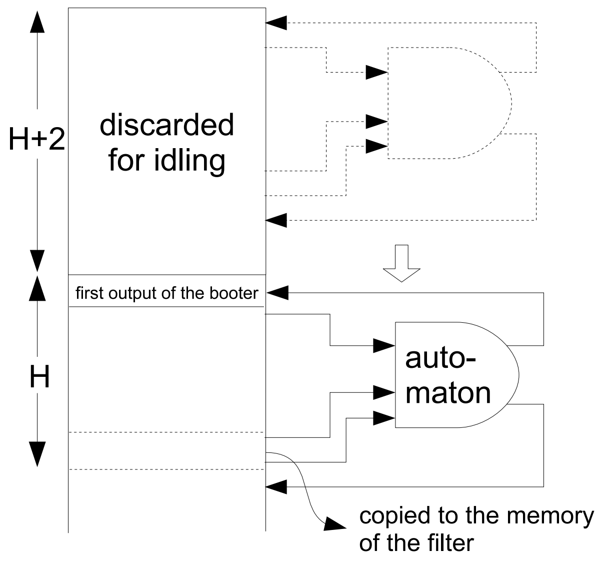
H+2
discarded for idling
first output of the booter
H
auto-maton
copied to the memory of the filter

Figure 8: After the Key and IV set-up.

to the shift register is written into the $(H + 1)$-st entry of the array. For the next generation, we shift the automaton downwards by one, and proceed in the same way.

For idling, we iterate this for $H + 2$ times. Then, the latest modified row in the array is the $(2H + 2)$-nd row, and it is copied to the 128-bit memory in the filter of CryptMT3. We discard the top $H + 2$ entries of the array. This completes the Key and IV set-up. Figure 8 shows the state after the set-up.

After the set-up, the booter produces 128-bit integer outputs, at most n times. Let L be the number of bits in the message. If $L \leq n \times 64$, then we do not need the mother generator. We generate the necessary number of 128-bit integers by the booter, and pass them to the filter to obtain the required outputs. If $L \geq n \times 64$, then, we generate n 128-bit integers by the booter, and pass them to the filter to obtain n 64-bit integers, which are used as the first outputs. At the same time, these n 128-bit integers are recorded in the array, and they are passed to SFMT as the initial state.

To eliminate the possibility of shorter period than $2^{19937} - 1$, we set the 32 MSBs of the first row of the state array of SFMT to the magic number `0x4d734e48` in the hexadecimal representation, as explained in §2. That is, we start the recursion (7) of SFMT with $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ being the array of length n generated by the booter (with 32 bits replaced with a magic constant), and then SFMT produces $\mathbf{x}_n, \mathbf{x}_{n+1}, \ldots$. Since $\mathbf{x}_n$ might be easier to guess because of the constant part in the initial state, we skip $\mathbf{x}_n$ and pass the 128-bit integers $\mathbf{x}_{n+1}, \mathbf{x}_{n+2}, \ldots$ to the filter.

The first outputs come from the booter. One may argue why not using the booter forever, without using the mother generator. The answer is that we do not need to care about the attacks to the booter based on a long output stream.

12 A Concrete Example Using 128-bit Instructions

Recent CPUs often have Single Instruction Multiple Data (SIMD) instructions. These instructions treat a quadruple of 32-bit integers at one time. We propose an LFSR and a uniform quasigroup filter, based on 128-bit instructions, named CryptMT Version 3 (CryptMT3) in the rest of this paper. CryptMT3 is one of the phase 3 candidates in eSTREAM stream cipher competition [25]. We shall describe the generation algorithm below.

12.1 SIMD-oriented Fast MT

In the LFSR (5), we assume that each x_i is a 128-bit integer or equivalently a vector in $\mathbb{F}_2^{128}$. We choose the following recursion: $n = 156$ and

$$\begin{aligned} x_{156+j} \quad := \quad & (x_{156+j-1} \ \& \ \texttt{ffdfafdf f5dabfff ffdbffff ef7bffff}) \oplus \\ & (x_{108+j} >>_{64} 3) \oplus x_{108+j}[2][0][3][1] \oplus (x_j[0][3][2][1]). \end{aligned} \tag{7}$$

Here, & denotes the bit-wise-and operation, and the hexadecimal integer is a constant 128-bit integer for the bit-mask. The notation $\oplus$ is bitwise exor. The notation

$$(x_{108+j} >>_{64} 3)$$

means that x_{108+j} is considered as two 64-bit integers, and each of them is shifted to the right by 3 bits. The notation $x_{108+j}[2][0][3][1]$ is a permutation of four 32-bit integers. The 128-bit integer x_{108+j} is considered as a quadruple of (0th, 1st, 2nd, 3rd) 32-bit integers, and then they are permuted by $2 \to 0, 1 \to 0, 2 \to 3, 3 \to 1$. The next notation $x_j[0][3][2][1]$ is a similar permutation. These instructions are available both in SSE2 SIMD instructions for Intel processors and in AltiVec SIMD instructions in PowerPC. We call this generator SIMD-oriented Fast MT (SFMT) (This is a variant of Chapter 1). A description is in Figure 9.

We proved its 155-dimensional equidistribution property. We proved that, if the third component $x_0[3]$ of x_0 is `0x4d734e48`, then the period of the generated sequence of the SFMT is a multiple of the Mersenne prime $2^{19937}-1$. Note that since 19937 is a prime, there is no intermediate field of $\mathbb{F}_{2^{19937}}$. This is in contrast to SNOW1.0, where the existence of the intermediate field $\mathbb{F}_{2^{32}}$ introduces some weakness (see [9]).

12.2 A Modified Multiplicative Filter

Our filter A_F has $I_F = S_F$ being the set of 128-bit integers, and O_F being the set of 64-bit integers, as described below.

For given 128-bit integers $y \in I_F$ and $x \in S_F$, we define

$$f_F(y, x) := (y \oplus (y[0][3][2][1] >>_{32} 1)) \tilde{\times}_{32} x. \tag{8}$$

Here, the notation "$>>_{32}$ 1" means to consider a 128-bit integer as a quadruple of 32-bit integers, and then shift each of them to the right by 1 bit. The binary operator $x \tilde{\times}_{32} y$ means that 32-bit wise binary operation $\tilde{\times}$ (see Example 10.3) is applied for each 32-bit components, namely, i-th 32-bit integer of $x \tilde{\times}_{32} y$ is $x[i] \tilde{\times} y[i]$ $(i = 0, 1, 2, 3)$.

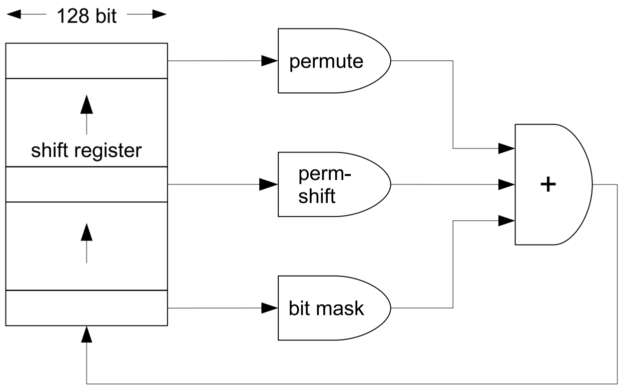

Figure 9: The mother generator of CryptMT3

SIMD-oriented Fast Mersenne Twister.
permute: $\mathbf{y} \mapsto \mathbf{y}[0][3][2][1]$.
perm-shift: $\mathbf{y} \mapsto \mathbf{y}[2][0][3][1] \oplus (\mathbf{y} >>_{64} 3)$.
bit-mask: `ffdfafdf f5dabfff ffdbffff ef7bffff`

The operation applied to y is an invertible linear transformation, hence is bijective. Since $\tilde{\times}$ is bi-bijective, so is f_F. The purpose to introduce the permutation-shift is to mix the information among four 32-bit memories in the filter, and to send the information of the upper bits to the lower bits. This supplements the multiplication, which lacks this direction of transfer of the information.

The output function is

$$o_F(y) := \mathrm{LSB}_{32}^{16}(y \oplus (y >>_{32} 16)). \tag{9}$$

This means that y is considered as a quadruple of 32-bit integers, and for each of them, we take the exor of the MSB 16 bits and LSB 16 bits. Thus we obtain four 16-bit integers, which is the output 64-bit integer (see Figure 10). To obtain 8-bit integers, we dissect it into 8 pieces.

CryptMT3 is the combination of the SIMD-oriented Fast MT (§12.1) and this filter. Initialization by the booter is explained in § 11.1.

For the period of CryptMT3, we have $Q = 2^{19937} - 1$ and $q \leq 2^{31} - 1$ and $k = 155$ $o_F : S_F \to O_F = \mathbb{F}_2^m$ is extracting 64 bits from the 128 bits, then $r = 2^{128-64}/2^{128} = 2^{-64}$. Assigning these numbers to the condition of Theorem 10.9:

$$r^{-(k+1)} > q \cdot (\#(S_F))^2,$$

we can say the output sequence of CryptMT3 has a period of non-zero multiple of $2^{19937} - 1$.

12.3 Speed Comparison

Comparison of the speed of generation for stream ciphers is a delicate problem: it depends on the platform, compilers, and so on. Here we compare the number of cycles consumed per byte, by CryptMT3, HC256, SOSEMANUK, Salsa20,

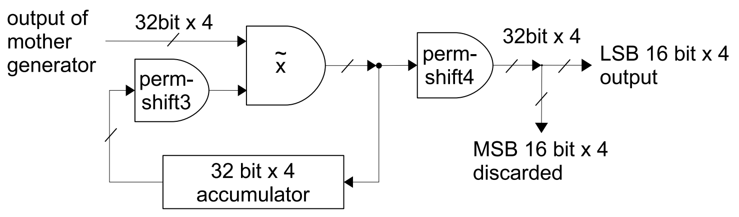

Figure 10: Filter of CryptMT3.

perm-shift3: $\mathbf{y} \mapsto \mathbf{y} \oplus (\mathbf{y}[0][3][2][1] >>_{32} 1)$.
perm-shift4: $\mathbf{y} \mapsto \mathbf{y} \oplus (\mathbf{y} >>_{32} 16)$.
$\tilde{\times}$: multiplication of 33-bit odd integers.

Dragon (these are the five candidates in eSTREAM software cipher phase 3 permitting 256-bit Key), SNOW2.0 and AES (counter-mode), in three different CPUs: Intel Core 2 Duo, AMD-Athlon X2, and Motorola PowerPC G4, using eSTREAM timing-tool [11]. The data are listed in Table 10. Actually, they are copied from Bernstein's page [2].

CryptMT3 is the fastest in generation in Intel Core 2 Duo CPU, reflecting the efficiency of SIMD operations in this newer CPU. CryptMT3 is slower in Motorola PowerPC. This is because AltiVec (SIMD of PowerPC) lacks 32-bit integer multiplication (so we used non-SIMD multiplication instead).

Table 10: Summary from eSTREAM benchmark [2]

	Core 2 Duo	Athlon 64 X2	PowerPC G4
Primitive	Stream	Stream	Stream
CryptMT3	2.95	4.73	9.23
HC-256	3.42	4.26	6.17
SOSEMANUK	3.67	4.41	6.17
SNOW-2.0	4.03	4.86	7.06
Salsa20	7.12	7.64	4.24
Dragon	7.61	8.11	8.39
AES-CTR	19.08	20.42	34.81

The number of cycles in Key set-up and IV set-up are listed in Table 11. The key set-up and IV set-up speed are less important than stream generation speed. Still, it is important for short message. HC-256 and SOSEMANUK are faster than CryptMT3 for Athlon and PowerPC, however, HC-256 is slower at IV set-up and SOSEMANUK is slower at key set-up.

13 Conclusions of Chapter 2

We proposed combination of an LFSR and a uniform quasigroup filter as a stream cipher in software. As a concrete example, we implemented CryptMT3 generator. CryptMT3 is as fast as SNOW2.0 and faster than AES counter-mode for recent CPUs. CryptMT3 satisfies the conditions of Theorem 10.9 and Propo-

Table 11: Summary from eSTREAM benchmark (setup cycle) [2]

	Core 2 Duo		Athlon 64 X2		PowerPC G4	
Primitive	Key setup	IV setup	Key	IV	Key	IV
CryptMT3	61.41	514.42	107	505.64	90.71	732.8
HC-256	61.31	83805.33	105.11	88726.2	87.71	71392
SOSEMANUK	848.51	624.99	1183.69	474.13	1797.03	590.47
SNOW-2.0	90.42	469.02	110.7	567	107.81	719.38
Salsa20	19.71	14.62	61.22	51.09	69.81	42.12
Dragon	121.42	1241.67	120.21	1469.43	134.6	1567.54
AES-CTR	625.44	18.9	905.65	50	305.81	34.11

sition 10.7, and it can be proved to have the astronomical period $\geq 2^{19937}-1$ and the 156-dimensional equidistribution property as a 64-bit integer generator (and hence 1241-dimensional equidistribution property as a 8-bit integer generator).

CryptMT3 uses integer multiplication instead of an S-box. This is an advantage over generators with large look-up tables for fast software implementation of the S-box, such as SNOW or AES, where cache-timing attacks might be applied [1].

A toy model of CryptMT3 shows high algebraic degrees and nonlinearity for the multiplicative filter, which supports its effectiveness. See §10.3 and §10.4.

Acknowledgements

The author would like to express his sincere gratitude to Professor Makoto Matsumoto for his enthusiastic advice and encouragement.

References

[1] D. J. Bernstein. Cache-timing attack on aes. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[2] D. J. Bernstein. Software timings. http://cr.yp.to/streamciphers/timings.html.

[3] R.P. Brent and P. Zimmermann. Random number generators with period divisible by a Mersenne prime. In *Computational Science and its Applications - ICCSA 2003*, volume 2667, pages 1–10, 2003.

[4] R.P. Brent and P. Zimmermann. Algorithms for finding almost irreducible and almost primitive trinomials. *Fields Inst. Commun.*, 41:91–102, 2004.

[5] N. Courtois. http://eprint.iacr.org/2005/243.

[6] N. Courtois. Fast algebraic attacks on stream ciphers with linear feedback. *In D. Boneh, editor, Advances in Cryptology - CRYPTO 2003*, pages 176–194, 2003.

[7] R. Couture, P. L'Ecuyer, and S. Tezuka. On the distribution of k-dimensional vectors for simple and combined Tausworthe sequences. *Math. Comp.*, 60(202):749–761, 1993.

[8] P. Ekdahl and T. Johansson. Snow-a new stream cipher. In *Proceedings of First Open NESSIE Workshop*. KU-Leuven, 2000.

[9] P. Ekdahl and T. Johansson. A new version of the stream cipher snow. In *Selected Areas in Cryptography, SAC 2002, LNCS 2595*, pages 47–61. Springer Verlag, 2002.

[10] Endianness from Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Endianness.

[11] estream – the ecrypt stream cipher project – phase 3. http://www.ecrypt.eu.org/stream/index.html.

[12] Sam Fuller. Motorola's AltiVec Technology. http://www.freescale.com/files/32bit/doc/fact_sheet/ALTIVECWP.pdf.

[13] M. Fushimi. Random number generation with the recursion $x_t = x_{t-3p} \oplus x_{t-3q}$. *Journal of Computational and Applied Mathematics*, 31:105–118, 1990.

[14] *Intel 64 and IA-32 Architectures Optimization Reference Manual*. http://www.intel.com/design/processor/manuals/248966.pdf.

[15] D. E. Knuth. *The Art of Computer Programming. Vol.2. Seminumerical Algorithms*. Addison-Wesley, Reading, Mass., 3rd edition, 1997.

[16] P. L'Ecuyer. A search for good multiple recursive random number genarators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, April 1993.

[17] P. L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Math. Comp.*, 65(213):203–213, 1996.

[18] P. L'Ecuyer. Tables of maximally equidistributed combined lfsr generators. *Math. Comp.*, 68(225):261–269, 1999.

[19] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 15(4):346–361, 2006.

[20] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[21] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, January 1998. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`.

[22] M. Matsumoto and T. Nishimura. A nonempirical test on the weight of pseudorandom number generators. In *Monte Carlo and Quasi-Monte Carlo methods 2000*, pages 381–395. Springer-Verlag, 2002.

[23] M. Matsumoto and T. Nishimura. Sum-discrepancy test on pseudorandom number generators. *Mathematics and Computers in Simulation*, 62(3-6):431–442, 2003.

[24] M. Matsumoto, M. Saito, T. Nishimura, and M. Hagita. Cryptanalysis of CryptMT: Effect of huge prime period and multiplicative filter. `http://www.ecrypt.eu.org/stream/cryptmtfubuki.html`.

[25] M. Matsumoto, M. Saito, T. Nishimura, and M Hagita. CryptMT stream cipher version 3. Submitted to eSTREAM stream cipher proposals, `http://www.ecrypt.eu.org/stream/cryptmtp3.html`.

[26] M. Matsumoto, M. Saito, T. Nishimura, and M Hagita. A fast stream cipher with huge state space and quasigroup filter for software. In *Selected Areas in Cryptography 2007*, pages 246–263. Springer Verlag, 2008.

[27] F. Panneton and P. L'Ecuyer. On the Xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation*, 15(4):346–361, 2005.

[28] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear reccurences modulo 2. *ACM Transactions on Mathematical Software*, 32(1):1–16, 2006.

[29] M. Saito and M. Matsumoto. SIMD-oriented fast mersenne twister:a 128-bit pseudorandom number generator. In *Monte Carlo and Quasi-Monte Carlo Methods 2006*, LNCS, pages 607–622. Springer, 2008.

[30] SIMD From Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/SIMD`.

Table 12: $k(v)$ and $d(v)$ (in parentheses) of SFMT as a 32-bit integer generator.

v	607	1279	2281	4253	11213	19937	44497	86243	132049	216091
1	606(1)	1274(5)	2278(3)	4253(0)	11207(6)	19937(0)	44490(7)	86239(4)	132048(1)	216089(2)
2	303(0)	638(1)	1139(1)	2125(1)	5604(2)	9966(2)	22247(1)	43121(0)	66024(0)	108043(2)
3	201(1)	425(1)	760(0)	1416(1)	3735(2)	6644(1)	14831(1)	28747(0)	44015(1)	72030(0)
4	151(0)	319(0)	570(0)	1062(1)	2802(1)	4982(2)	11123(1)	21560(0)	33011(1)	54021(1)
5	121(0)	255(0)	455(1)	850(0)	2242(0)	3985(2)	8898(1)	17247(1)	26409(0)	40536(2682)
6	100(1)	212(1)	379(1)	708(0)	1868(0)	3322(0)	7415(1)	14373(0)	22007(1)	36014(1)
7	86(0)	182(0)	325(0)	606(1)	1601(0)	2847(1)	6200(156)	12320(0)	16513(2351)	30870(0)
8	75(0)	159(0)	284(1)	531(0)	1401(0)	2491(1)	5561(1)	10780(0)	16505(1)	27011(0)
9	62(5)	141(1)	252(1)	440(32)	1058(187)	2214(1)	4178(766)	9364(218)	12400(2272)	20268(3742)
10	60(0)	127(0)	225(3)	408(17)	1058(63)	1993(0)	4176(273)	8088(536)	12389(815)	20268(1341)
11	55(0)	115(1)	206(1)	372(14)	1019(0)	1812(0)	2947(1098)	5399(2441)	8325(3679)	19584(60)
12	44(6)	102(4)	185(5)	354(0)	934(0)	1544(117)	2803(905)	5399(1787)	8264(2740)	17788(219)
13	41(5)	88(10)	167(8)	268(59)	783(79)	1248(285)	2803(619)	5392(1242)	8264(1893)	16037(585)
14	40(3)	80(11)	140(22)	268(35)	714(86)	1248(176)	1393(1785)	5392(768)	8252(1180)	13508(1927)
15	40(0)	76(9)	140(12)	268(15)	701(46)	1244(85)	1393(1573)	2696(3053)	8252(551)	13508(898)
16	37(0)	76(3)	140(2)	265(0)	700(0)	1244(2)	1393(1388)	2696(2694)	8252(1)	13505(0)
17	25(10)	45(30)	96(38)	141(109)	512(147)	629(543)	1392(1225)	2696(2377)	4133(3634)	6761(5950)
18	21(12)	45(26)	77(49)	140(96)	357(265)	629(478)	1392(1080)	2696(2095)	4133(3203)	6761(5244)
19	21(10)	40(27)	77(43)	140(83)	353(237)	624(425)	1392(949)	2696(1843)	4132(2817)	6761(4612)
20	21(9)	40(23)	72(42)	140(72)	353(207)	624(372)	1392(832)	2696(1616)	4128(2474)	6756(4048)
21	21(7)	40(20)	72(36)	136(66)	353(180)	624(325)	1392(726)	2696(1410)	4128(2160)	6756(3534)
22	21(6)	40(18)	72(31)	136(57)	353(156)	624(282)	1392(630)	2696(1224)	4128(1874)	6756(3066)
23	21(5)	40(15)	72(27)	136(48)	352(135)	624(242)	1392(542)	2696(1053)	4128(1613)	6756(2639)
24	20(5)	40(13)	72(23)	136(41)	352(115)	624(206)	1392(462)	2696(897)	4128(1374)	6756(2247)
25	20(4)	40(11)	72(19)	136(34)	352(96)	624(173)	1392(387)	2696(753)	4128(1153)	6756(1887)
26	20(3)	40(9)	72(15)	136(27)	352(79)	624(142)	1392(319)	2696(621)	4128(950)	6756(1555)
27	20(2)	40(7)	72(12)	136(21)	352(63)	624(114)	1392(256)	2696(498)	4128(762)	6756(1247)
28	20(1)	40(5)	72(9)	136(15)	352(48)	624(88)	1392(197)	2696(384)	4128(588)	6756(961)
29	20(0)	40(4)	72(6)	136(10)	352(34)	624(63)	1392(142)	2696(277)	4128(425)	6756(695)
30	20(0)	40(2)	72(4)	136(5)	352(21)	624(40)	1392(91)	2696(178)	4128(273)	6756(447)
31	19(0)	40(1)	72(1)	136(1)	352(9)	624(19)	1392(43)	2696(86)	4128(131)	6756(214)
32	18(0)	39(0)	71(0)	132(0)	350(0)	622(1)	1390(0)	2695(0)	4126(0)	6752(0)
Δ	96	258	416	861	2264	4188	16457	28056	38918	49806

Table 13: $k(v)$ and $d(v)$ (in parentheses) of SFMT as a 64-bit integer generator. (for v is 33 to 64)

v	607	1279	2281	4253	11213	19937	44497	86243	132049	216091
1	603(4)	1278(1)	2280(1)	4253(0)	11211(2)	19937(0)	44496(1)	86242(1)	132049(0)	216087(4)
2	303(0)	639(0)	1140(0)	2125(1)	5606(0)	9968(0)	22247(1)	43119(2)	66024(0)	108045(0)
3	199(3)	426(0)	759(1)	1417(0)	3737(0)	6645(0)	14832(0)	28746(1)	44016(0)	72029(1)
4	151(0)	319(0)	570(0)	1062(1)	2802(1)	4983(1)	11124(0)	21560(0)	33012(0)	54022(0)
5	121(0)	254(1)	455(1)	850(0)	2242(0)	3987(0)	8898(1)	17248(0)	26409(0)	43218(0)
6	100(1)	212(1)	379(1)	708(0)	1868(0)	3321(1)	7415(1)	14373(0)	22008(0)	36015(0)
7	86(0)	182(0)	325(0)	607(0)	1601(0)	2847(1)	6356(0)	12319(1)	18864(0)	30869(1)
8	75(0)	159(0)	285(0)	530(1)	1401(0)	2491(1)	5561(1)	10780(0)	16505(1)	27011(0)
9	67(0)	141(1)	253(0)	472(0)	1245(0)	2214(1)	4943(1)	5392(4190)	14671(1)	16887(7123)
10	60(0)	127(0)	227(1)	404(21)	1121(0)	1993(0)	2089(2360)	5392(3232)	13081(123)	16887(4722)
11	51(4)	101(15)	207(0)	386(0)	529(490)	1812(0)	2089(1956)	5392(2448)	11359(645)	16887(2757)
12	50(0)	101(5)	190(0)	354(0)	529(405)	1556(105)	2089(1619)	5392(1794)	8254(2750)	16887(1120)
13	31(15)	98(0)	160(15)	134(193)	529(333)	934(599)	2089(1333)	5392(1242)	8254(1903)	16619(3)
14	31(12)	90(1)	70(92)	134(169)	529(271)	934(490)	2089(1089)	5392(768)	4126(5306)	12255(3180)
15	31(9)	85(0)	70(82)	134(149)	529(218)	622(707)	2089(877)	1348(4401)	4126(4677)	10135(4271)
16	31(6)	77(2)	70(72)	134(131)	529(171)	622(624)	2089(692)	1348(4042)	4126(4127)	10135(3370)
17	31(4)	40(35)	70(64)	134(116)	529(130)	622(550)	1397(1220)	1348(3725)	4126(3641)	10135(2576)
18	29(4)	40(31)	70(56)	134(102)	353(269)	622(485)	1397(1075)	1348(3443)	4126(3210)	6756(5249)
19	21(10)	40(27)	70(50)	134(89)	353(237)	312(737)	1397(944)	1348(3191)	4126(2823)	6756(4617)
20	21(9)	40(23)	36(78)	134(78)	353(207)	312(684)	1397(827)	1348(2964)	2064(4538)	6756(4048)
21	21(7)	40(20)	36(72)	68(134)	353(180)	312(637)	1397(721)	1348(2758)	2064(4224)	6756(3534)
22	21(6)	40(18)	36(67)	68(125)	353(156)	312(594)	1397(625)	1348(2572)	2064(3938)	6756(3066)
23	21(5)	40(15)	36(63)	68(116)	353(134)	312(554)	1393(541)	1348(2401)	2064(3677)	6756(2639)
24	21(4)	40(13)	36(59)	68(109)	353(114)	312(518)	1393(461)	1348(2245)	2064(3438)	6756(2247)
25	10(14)	20(31)	36(55)	68(102)	176(272)	312(485)	696(1083)	1348(2101)	2064(3217)	3378(5265)
26	10(13)	20(29)	36(51)	68(95)	176(255)	312(454)	696(1015)	1348(1969)	2064(3014)	3378(4933)
27	10(12)	20(27)	36(48)	68(89)	176(239)	312(426)	696(952)	1348(1846)	2064(2826)	3378(4625)
28	10(11)	20(25)	36(45)	68(83)	176(224)	312(400)	696(893)	1348(1732)	2064(2652)	3378(4339)
29	10(10)	20(24)	36(42)	68(78)	176(210)	312(375)	696(838)	1348(1625)	2064(2489)	3378(4073)
30	10(10)	20(22)	36(40)	68(73)	176(197)	312(352)	696(787)	1348(1526)	2064(2337)	3378(3825)
31	10(9)	20(21)	36(37)	68(69)	176(185)	312(331)	696(739)	1348(1434)	2064(2195)	3378(3592)
32	10(8)	20(19)	36(35)	68(64)	176(174)	312(311)	696(694)	1348(1347)	2064(2062)	3378(3374)

Table 14: $k(v)$ and $d(v)$ (in parentheses) of SFMT as a 64-bit integer generator. (for v is 33 to 64)

v	607	1279	2281	4253	11213	19937	44497	86243	132049	216091
33	10(8)	20(18)	36(33)	68(60)	176(163)	312(292)	696(652)	1348(1265)	2064(1937)	3378(3170)
34	10(7)	20(17)	36(31)	68(57)	176(153)	312(274)	696(612)	1348(1188)	2064(1819)	3378(2977)
35	10(7)	20(16)	36(29)	68(53)	176(144)	312(257)	696(575)	1348(1116)	2064(1708)	3378(2796)
36	10(6)	20(15)	36(27)	68(50)	176(135)	312(241)	696(540)	1348(1047)	2064(1604)	3378(2624)
37	10(6)	20(14)	36(25)	68(46)	176(127)	312(226)	696(506)	1348(982)	2064(1504)	3378(2462)
38	10(5)	20(13)	36(24)	68(43)	176(119)	312(212)	696(474)	1348(921)	2064(1410)	3378(2308)
39	10(5)	20(12)	36(22)	68(41)	176(111)	312(199)	696(444)	1348(863)	2064(1321)	3378(2162)
40	10(5)	20(11)	36(21)	68(38)	176(104)	312(186)	696(416)	1348(808)	2064(1237)	3378(2024)
41	10(4)	20(11)	36(19)	68(35)	176(97)	312(174)	696(389)	1348(755)	2064(1156)	3378(1892)
42	10(4)	20(10)	36(18)	68(33)	176(90)	312(162)	696(363)	1348(705)	2064(1080)	3378(1767)
43	10(4)	20(9)	36(17)	68(30)	176(84)	312(151)	696(338)	1348(657)	2064(1006)	3378(1647)
44	10(3)	20(9)	36(15)	68(28)	176(78)	312(141)	696(315)	1348(612)	2064(937)	3378(1533)
45	10(3)	20(8)	36(14)	68(26)	176(73)	312(131)	696(292)	1348(568)	2064(870)	3378(1424)
46	10(3)	20(7)	36(13)	68(24)	176(67)	312(121)	696(271)	1348(526)	2064(806)	3378(1319)
47	10(2)	20(7)	36(12)	68(22)	176(62)	312(112)	696(250)	1348(486)	2064(745)	3378(1219)
48	10(2)	20(6)	36(11)	68(20)	176(57)	312(103)	696(231)	1348(448)	2064(687)	3378(1123)
49	10(2)	20(6)	36(10)	68(18)	176(52)	312(94)	696(212)	1348(412)	2064(630)	3378(1032)
50	10(2)	20(5)	36(9)	68(17)	176(48)	312(86)	696(193)	1348(376)	2064(576)	3378(943)
51	10(1)	20(5)	36(8)	68(15)	176(43)	312(78)	696(176)	1348(343)	2064(525)	3378(859)
52	10(1)	20(4)	36(7)	68(13)	176(39)	312(71)	696(159)	1348(310)	2064(475)	3378(777)
53	10(1)	20(4)	36(7)	68(12)	176(35)	312(64)	696(143)	1348(279)	2064(427)	3378(699)
54	10(1)	20(3)	36(6)	68(10)	176(31)	312(57)	696(128)	1348(249)	2064(381)	3378(623)
55	10(1)	20(3)	36(5)	68(9)	176(27)	312(50)	696(113)	1348(220)	2064(336)	3378(550)
56	10(0)	20(2)	36(4)	68(7)	176(24)	312(44)	696(98)	1348(192)	2064(294)	3378(480)
57	10(0)	20(2)	36(4)	68(6)	176(20)	312(37)	696(84)	1348(165)	2064(252)	3378(413)
58	10(0)	20(2)	36(3)	68(5)	176(17)	312(31)	696(71)	1348(138)	2064(212)	3378(347)
59	10(0)	20(1)	36(2)	68(4)	176(14)	312(25)	696(58)	1348(113)	2064(174)	3378(284)
60	10(0)	20(1)	36(2)	68(2)	176(10)	312(20)	696(45)	1348(89)	2064(136)	3378(223)
61	9(0)	20(0)	36(1)	68(1)	176(7)	312(14)	696(33)	1348(65)	2064(100)	3378(164)
62	9(0)	20(0)	36(0)	68(0)	176(4)	312(9)	696(21)	1348(43)	2064(65)	3378(107)
63	9(0)	19(1)	36(0)	67(0)	176(1)	312(4)	696(10)	1348(20)	2064(32)	3378(52)
64	9(0)	19(0)	35(0)	66(0)	175(0)	311(0)	695(0)	1347(0)	2063(0)	3376(0)
Δ	273	629	1527	2913	7110	14089	31559	74962	94256	128554

公表論文

(1) SIMD-oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator,
Mutsuo Saito and Makoto Matsumoto,
Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, (2008) 607-622.

(2) A Fast Stream Cipher with Huge State Space and Quasigroup Filter for Software,
Matsumoto, M., Saito, M., Nishimura, T. and Hagita, M.
Selected Areas in Cryptography 2007, Lecture Notes in Computer Science (LNCS), vol. 4876, (2007) 246-263.