CryptMT Stream Cipher Ver. 3: Description ⋆

Makoto Matsumoto[1], Mutsuo Saito[2], Takuji Nishimura[3], and Mariko Hagita[4]

[1] Dept. of Math., Hiroshima University, m-mat@math.sci.hiroshima-u.ac.jp
[2] Dept. of Math., Hiroshima University, saito@math.sci.hiroshima-u.ac.jp
[3] Dept. of Math. Sci., Yamagata University, nisimura@sci.kj.yamagata-u.ac.jp
[4] Dept. of Info. Sci., Ochanomizu University, hagita@is.ocha.ac.jp

Abstract. CryptMT Version 3 (CryptMT3) is a stream cipher obtained by combining a large LFSR and a nonlinear filter with memory using integer multiplication. Its period is proved to be no less than $2^{19937}-1$, and the 8-bit output sequence is at least 1241-dimensionally equidistributed. It is one of the fastest stream ciphers on a CPU with SIMD operations, such as Intel Core 2 Duo.

1 Introduction

In this article, we discuss pseudorandom number generators (PRNGs) for stream ciphers. We assume that the PRNG is implemented in software, and the platform is a 32-bit CPU with enough memory and fast integer multiplication.

Our proposal [5][6] is to combine a huge state linear generator (called the mother generator) and a filter with memory, as shown in Figure 1.

Definition 1. *(Generator with a filter with memory.) Let X be a finite set (typically the set of the word-size integers). The mother generator G generates a sequence $x_0, x_1, x_2, \ldots \in X$. Let Y be a finite set, which is the set of the possible states of the memory in the filter. We take a $y_0 \in Y$. Let $f : Y \times X \to Y$ be the state transition function of the memory of the filter, that is, the content y_i of the memory is changed by the recursion*

$$y_{i+1} := f(y_i, x_i).$$

The output at the i-th step is given by $g(y_i)$, where $g : Y \to O$ is the output function which converts the content of the memory to an output symbol in O.

⋆ CryptMT is proposed to eSTREAM Project http://www.ecrypt.eu.org/stream/. The reference codes are available there. This work is supported in part by JSPS Grant-In-Aid #16204002, #18654021, #18740044, #19204002 and JSPS Core-to-Core Program No.18005.

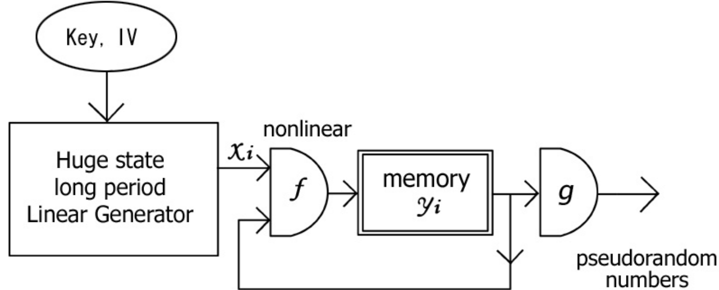

Fig. 1. Combined generator = linear generator + filter with memory.

In a previous manuscript [5], we chose the mother generator to be Mersenne Twister [4], which generates a sequence of 32-bit integers by an $\mathbb{F}_2$-linear recursion. The filter is given by

$$f(y, x) := y \times (x|1) \bmod 2^{32},\ \ g(y) := 8 \text{ MSBs of } y \tag{1}$$

where $(x|1)$ denotes x with LSB set to 1, and 8 MSBs mean the 8 most significant bits of the integer y. Initially, the memory is set to an odd integer y_0. This is CryptMT Version 1 (CryptMT1). There has been no attacks reported to this generator (even non-practical attacks). We introduced CryptMT Version 2 [7] and Version 3 [8], not to improve the security, but to improve the speed of initialization and generation. This manuscript is based on [8]. Theoretical analysis of this type of generators is developed in [9], where the quasigroup property of the filter plays the role of “balanced filter”.

2 CryptMT Version 3: A new variant based on 128-bit operations

Modern CPUs often have single-instruction-multiple-data (SIMD) operations. Typically, a quadruple of 32-bit registers is considered as a single 128-bit register. CryptMT Ver.3 proposed here is a modification of the Version 1, so that it fits to the high-speed SIMD operations.

2.1 Notation

Let us fix the notations for 128-bit integers. A bold italic letter $\mathbf{x}$ denotes a 128-bit integer. It is a concatenation of four 32-bit registers, each of which is denoted by $\mathbf{x}[3], \mathbf{x}[2], \mathbf{x}[1], \mathbf{x}[0]$, respectively, from MSB to LSB.

The notation $\mathbf{x}[3][2]$ denotes the 64-bit integer obtained by concatenating the two 32-bit integers $\mathbf{x}[3]$ and $\mathbf{x}[2]$, in this order. Similarly, $\mathbf{x}[0][3][2][1]$ denotes the 128-bit integer obtained by permuting (actually rotating) the four 32-bit integers in $\mathbf{x}$. Thus, for example, $\mathbf{x} = \mathbf{x}[3][2][1][0]$ holds.

An operation on 128-bit registers that is executed for each 32-bit integer is denoted with the subscript 32. For example,

$$\mathbf{x} +_{32} \mathbf{y} := [(\mathbf{x}[3] + \mathbf{y}[3]), (\mathbf{x}[2] + \mathbf{y}[2]), (\mathbf{x}[1] + \mathbf{y}[1]), (\mathbf{x}[0] + \mathbf{y}[0])],$$

that is, the first 32-bit part is the addition of $\mathbf{x}[3]$ and $\mathbf{y}[3]$ modulo 2^{32}, the second 32-bit is that of $\mathbf{x}[2]$ and $\mathbf{y}[2]$ (without the carry from the second 32-bit part to the first 32-bit part, differently from the addition of 128-bit integers). The outer most [ ] in the right hand side is to emphasize that they are concatenated to give a 128-bit integer.

Similarly, for an integer S,

$$\mathbf{x} >>_{32} S := [(\mathbf{x}[3] >> S), (\mathbf{x}[2] >> S), (\mathbf{x}[1] >> S), (\mathbf{x}[0] >> S)]$$

means the shift right by S bits applied to each of the four 32-bit integers, and

$$\mathbf{x} >>_{64} S := [(\mathbf{x}[3][2] >> S), (\mathbf{x}[1][0] >> S)]$$

means the shifts applied to each of the two 64-bit integers.

In the following, we often use functions such as

$$\mathbf{x} \mapsto \mathbf{x} \oplus (\mathbf{x}[2][1][0][3] >>_{32} S),$$

which we call *perm-shift*. Here $\oplus$ means the bit-wise exclusive-or. The permutation [2][1][0][3] may be an arbitrary permutation, and the shift may be to the left. A function of the form

$$\mathbf{x} \mapsto \mathbf{x}[i_3][i_2][i_1][i_0] \oplus (\mathbf{x} >>_{32} S)$$

is also called a perm-shift, where $i_3 i_2 i_1 i_0$ is a permutation of $3, 2, 1, 0$. A perm-shift is an $\mathbb{F}_2$-linear transformation, and if $S \geq 1$ then it is a bijection. (Since its representation matrix is an invertible triangular matrix times a permutation matrix, under a suitable choice of the basis.)

Let n be a positive integer, and x be a 32-bit integer. The n most significant bits of x are denoted by $\mathrm{MSB}^n(x)$. Similar notation $\mathrm{LSB}^n(x)$ is also used. For a 128-bit integer $\mathbf{x}$, we define

$$\mathrm{MSB}^n_{32}(\mathbf{x}) := [\mathrm{MSB}^n(\mathbf{x}[3]), \mathrm{MSB}^n(\mathbf{x}[2]), \mathrm{MSB}^n(\mathbf{x}[1]), \mathrm{MSB}^n(\mathbf{x}[0])],$$

which is a $(n \times 4)$-bit integer.

A function $f : Y \times X \to Z$ is *bi-bijective* if for any fixed $x \in X$, the mapping $Y \to Z,\ y \mapsto f(y, x)$ is bijective, and for any fixed $y \in Y$, the mapping $X \to Z,\ x \mapsto f(y, x)$ is bijective. It is necessary that the cardinalities coincide: $\#(X) = \#(Y) = \#(Z)$.

2.2 SIMD Fast MT

The Version 3 adopts the following mother generator, named SIMD-oriented Fast Mersenne Twister (SFMT) [10].

Let N be an integer, and $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{N-1}$ be N 128-bit integers given as the initial state. A version of SFMT used here is to generate a sequence of 128-bit integers by the following $\mathbb{F}_2$-linear recursion:

$$\begin{aligned}\mathbf{x}_{N+j} := {} & (\mathbf{x}_{N+j-1}\ \&\ \text{128-bit MASK}) \oplus \\ & (\mathbf{x}_{M+j} >>_{64} S) \oplus (\mathbf{x}_{M+j}[2][0][3][1]) \oplus (\mathbf{x}_j[0][3][2][1]).\end{aligned} \quad (2)$$

Here, & denotes the bit-wise-and operation, so the first term is the result of the bit-mask of $\mathbf{x}_{N+j-1}$ by a constant 128-bit MASK. The second term is the concatenation of two 64-bit integers $(\mathbf{x}_{M+j}[3][2] >> S)$ and $(\mathbf{x}_{M+j}[1][0] >> S)$, as explained above. The third term is a permutation of four 32-bit integers in $\mathbf{x}_{M+j}$, and the last term is a rotation of those in $\mathbf{x}_j$. Thus, the SFMT is based on the N-th order linear recursion over the 128-dimensional vectors ${\mathbb{F}_2}^{128}$. Figure 2 describes the SFMT. By a computer search, we found the parameters $N = 156, M = 108, S = 3$ and MASK = `ffdfafdf f5dabfff ffdbffff ef7bffff` in the hexa-decimal notation. Such a mask is necessary to break the symmetry (i.e., without such asymmetry, if each 128-bit integer $\mathbf{x}_i$ in the initial state array satisfies $\mathbf{x}_i[3] = \mathbf{x}_i[2] = \mathbf{x}_i[1] = \mathbf{x}_i[0]$, then this equality holds ever after). We selected a mask with more 1's than 0's, so that we do not lose the information so much.

We proved that, if $\mathbf{x}_0[3] =$ `0x4d734e48`, then the period of the generated sequence of the SFMT is a multiple of the Mersenne prime $2^{19937} - 1$, and the output is 155-dimensionally equidistributed, using the method described in [10].

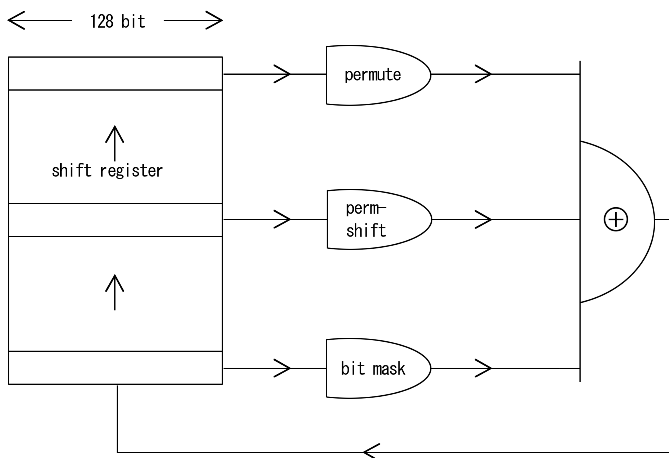

Fig. 2. The mother generator: SIMD Fast Mersenne Twister.
permute: $\mathbf{y} \mapsto \mathbf{y}[0][3][2][1]$.
perm-shift: $\mathbf{y} \mapsto \mathbf{y}[2][0][3][1] \oplus (\mathbf{y} >>_{64} 3)$.
bit-mask: `ffdfafdf f5dabfff ffdbffff ef7bffff`

These operations are chosen to fit SIMD instructions in modern CPUs such as Intel Core 2 Duo. We note that even for CPUs without SIMD, computation of such a recurring formula is fast since it fits the pipeline processing.

2.3 A new filter

The previously proposed filter (1) uses integer multiplication in the ring $\mathbb{Z}/2^{32}\mathbb{Z}$. To avoid the degenerations, we restrict the multiplication to the set of odd integers in $\mathbb{Z}/2^{32}\mathbb{Z}$, by setting the LSB to be 1 in (1).

In Version 3, we use the following binary operation $\tilde{\times}$ on $\mathbb{Z}/2^{32}\mathbb{Z}$ instead of $\times$: for $x, y \in \mathbb{Z}/2^{32}\mathbb{Z}$, we define

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

which is essentially the multiplication of 33-bit odd integers. Let S be the set of odd integers in $\mathbb{Z}/2^{33}\mathbb{Z}$. By regarding $\mathbb{Z}/2^{32}\mathbb{Z} = \{0, 1, \ldots, 2^{32} - 1\}$, we have a bijection

$$\varphi : \mathbb{Z}/2^{32}\mathbb{Z} \to S, \quad x \mapsto 2x + 1.$$

Then, $\tilde{\times}$ above is defined by

$$x \tilde{\times} y := \varphi^{-1}(\varphi(x) \times \varphi(y)),$$

where $\times$ denotes the multiplication in S. Thus, $\tilde{\times}$ is given by looking at the upper 32 bits of multiplications in S. Consequently, $\tilde{\times}$ is bi-bijective.

Most of modern CPUs have 32-bit integer multiplication but not 64-bit nor 128-bit multiplication. Thus, a simplest parallelization of (1) would be the following: $X = Y = (\mathbb{Z}/2^{32}\mathbb{Z})^4$, and

$$f(\mathbf{y}, \mathbf{x}) := \mathbf{y} \tilde{\times}_{32} \mathbf{x},$$

(that is, $f(\mathbf{y}, \mathbf{x})[i] := \mathbf{y}[i] \tilde{\times} \mathbf{x}[i]$ for $i = 3, 2, 1, 0$), and

$$g(\mathbf{y}) := \mathrm{MSB}_{32}^{8}(\mathbf{x})$$

is the output of (8×4)-bit integers (for notations, see §2.1).

In Version 3, we adopted a modified filter (see Figure 3) as follows. For a given pair of 128-bit integers $\mathbf{x}, \mathbf{y}$, we define

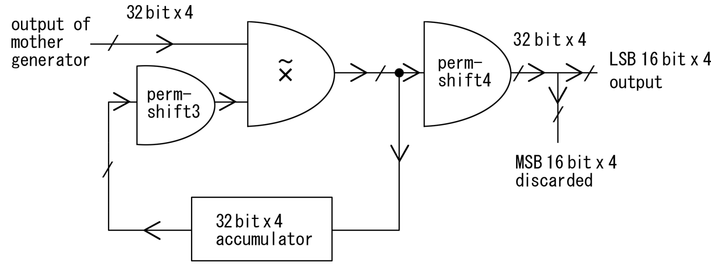

Fig. 3. Filter of CryptMT Ver.3.
perm-shift3: $\mathbf{y} \mapsto \mathbf{y} \oplus (\mathbf{y}[0][3][2][1] >>_{32} 1)$.
perm-shift4: $\mathbf{y} \mapsto \mathbf{y} \oplus (\mathbf{y} >>_{32} 16)$.
$\tilde{\times}$: multiplication of 33-bit odd integers.

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

The operation applied to $\mathbf{y}$ in the right hand side is a perm-shift (see §2.1), hence is bijective. Since $\tilde{\times}$ is bi-bijective, so is f. The purpose to introduce the perm-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

$$g(\mathbf{y}) := \mathrm{LSB}_{32}^{16}(\mathbf{y} \oplus (\mathbf{y} >>_{32} 16)). \tag{4}$$

Thus, the new filter has 128-bit of memory, receives a 128-bit integer, and output a (16 × 4)-bit integer. The compression ratio of this filter is (128:64), which is smaller than (32:8) in the previously proposed filter. This change of the ratio is for the speed, but might weaken the security. To compensate this, the output function takes the exclusive-or of the 16 MSBs and the 16 LSBs of $\mathbf{y}[i]$, $i = 3, 2, 1, 0$.

2.4 Conversion to 8-bit integers

Since the outputs of the filter are (16 × 4)-bit integers and the specification required is 8-bit integer outputs, we need to dissect them into 8-bit integers. Because of the nature of the 128-bit SIMD instructions, the following strategy is adopted for the speed. Let

$$\mathtt{LOWER16} := (\mathtt{0x0000ffff}, \mathtt{0x0000ffff}, \mathtt{0x0000ffff}, \mathtt{0x0000ffff})$$
$$\mathtt{UPPER16} := (\mathtt{0xffff0000}, \mathtt{0xffff0000}, \mathtt{0xffff0000}, \mathtt{0xffff0000})$$

be the 128-bit masks.

Let $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{2i}, \mathbf{y}_{2i+1}, \ldots$ be the content of the memory in the filter at every step, i.e., generated by $\mathbf{y}_{i+1} := f(\mathbf{y}_i, \mathbf{x}_i)$ in (3). Then, $\mathbf{y}_{2i}$ and $\mathbf{y}_{2i+1}$ are used to generate the i-th 128-bit integer output $\mathbf{z}_i$, by the formula

$$\mathbf{z}_i := [(\mathbf{y}_{2i} \oplus (\mathbf{y}_{2i} >>_{32} 16)) \& \mathtt{LOWER16}] \mid [(\mathbf{y}_{2i+1} \oplus (\mathbf{y}_{2i+1} <<_{32} 16)) \& \mathtt{UPPER16}]$$

where | denotes the bit-wise-or. Then, $\mathbf{z}_i$ is separated into 16 of 8-bit integers from the lower bits to the upper bits, and used as the 8-bit integer outputs.

2.5 A new booter for the initialization

SFMT in §2.2 requires $N = 156$ of 128-bit integers as the initial state. We need to expand the key and IV to an initial state at the initialization, but

this is expensive when the message length is much less than $N \times 128$ bits. Our strategy introduced in [7] is to use a smaller PRNG called the booter. Its role is to expand the key and IV to a sequence of 128-bit integers. The output of the booter is passed to the filter discussed above to generate the pseudorandom integer stream, and at the same time, used to fill the state of SFMT. Once the state of SFMT is filled up, then the generation is switched from the booter to SFMT.

The booter we adopted here is described in Figure 4. We choose an

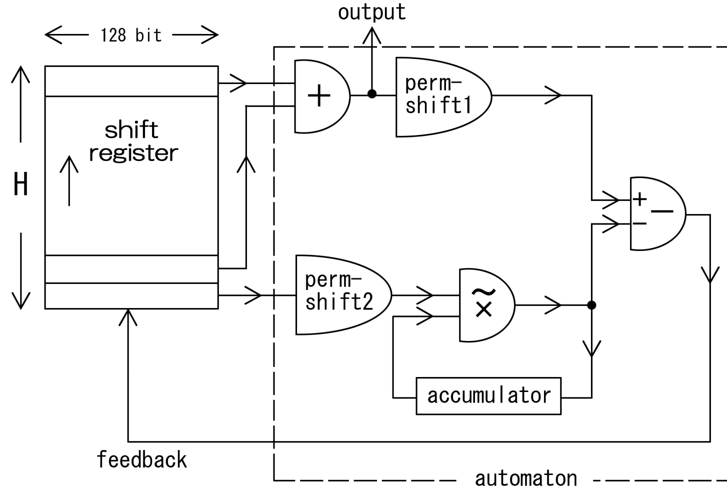

Fig. 4. Booter of CryptMT Ver.3.
`perm-shift1`: $\mathbf{x} \mapsto (\mathbf{x}[2][1][0][3]) \oplus (\mathbf{x} >>_{32} 13)$.
`perm-shift2`: $\mathbf{x} \mapsto (\mathbf{x}[1][0][2][3]) \oplus (\mathbf{x} >>_{32} 11)$.
$\tilde{\times}$: multiplication of (a quadruple of) 33-bit odd integers.

integer H later in §2.6 according to the sizes of the Key and IV. 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

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

Similarly to the notation $+_{32}$ (§2.1), $-_{32}$ denotes the subtraction modulo 2^{32} for each of the four 32-bit integers. The output of the j-th step is $\mathbf{x}_j +_{32} \mathbf{x}_{H+j-2}$.

As described in Figure 4, the booter consists of an automaton with three inputs and two outputs of 128-bit integers, with a shift register. In the implementation, the shift register is taken in an array of 128-bit integers with the length $2H+2+N$. This redundancy of the length is for the idling, as explained below.

2.6 Key and IV set-up

We assume that both the IV and the Key are given as arrays of 128-bit integers, of length from 1 to 16 for each. Thus, the Key-size can flexibly be 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.

In the set-up of the IV and the Key, these arrays are concatenated and copied twice to an array, as described in Figure 5. To break the symmetry,

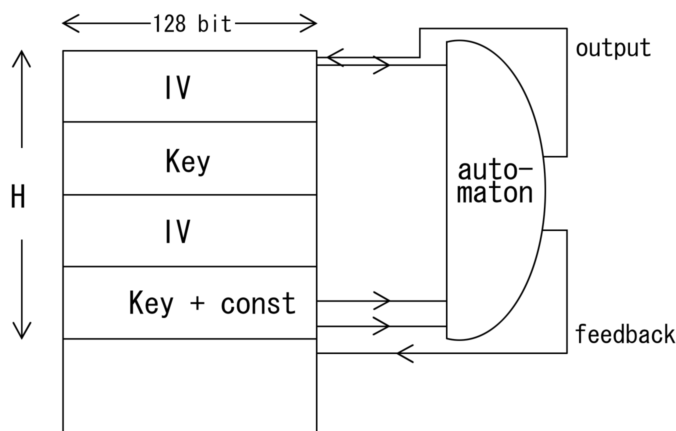

Fig. 5. Key and IV set-up. The IV-array and Key-array are concatenated and copied to an array twice. Then, a constant is added to the bottom of the second copy of the key to break a possible symmetry. The automaton is described in Figure 4.

we add a constant 128-bit integer $(846264, 979323, 265358, 314159)$ (denoting four 32-bit integers in a decimal notation) to the bottom row of the

second copy of the key (add means $+_{32}$ modulo 2^{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 4 is equipped on this array, as shown in Figure 5. 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 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. We discard the top $H + 2$ entries of the array. This completes the Key and IV set-up. Figure 6 shows the state after the set-up.

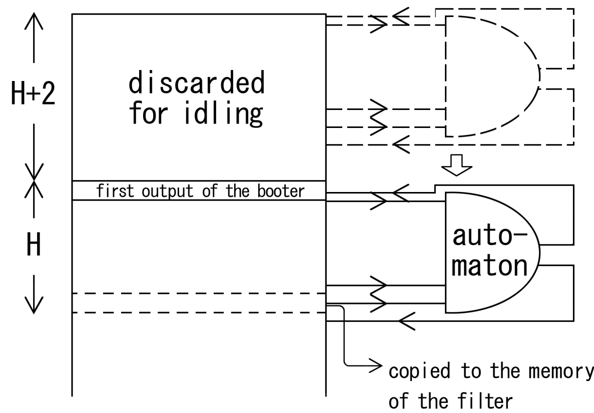

Fig. 6. After the Key and IV set-up.

After the set-up, the booter produces 128-bit integer outputs at most N times. Let L be the number of 8-bit integers in the message. If $L \times 8 \leq N \times 64$, 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 $L \times 8 \geq N \times 64$, 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 §2.2. This is illustrated in Figure 7. That is, we start the recursion (2)

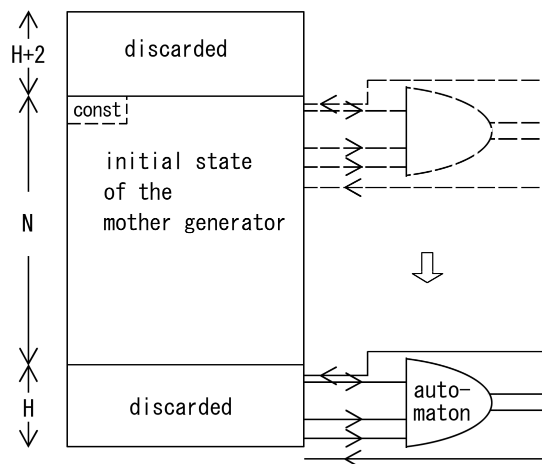

Fig. 7. Initialization of the SFMT mother generator.

of SFMT with $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{N-1}$ being the array of length N indicated in Figure 7, and 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 it and pass the 128-bit integers $\mathbf{x}_{N+1}, \mathbf{x}_{N+2}, \ldots$ to the filter.

3 Resistance of CryptMT Ver.3 to Standard Attacks

The cryptanalysis developed in §4 in [6] for CryptMT is also valid for the Version 3. We list some properties of the SFMT (§2.2) required in the following cryptanalysis. Algorithms to check these are described in [10].

Proposition 1. *SFMT is an automaton with the state space S being an array of 128-bit integers of the length 156 (hence having* $19968 = 128 \times 156$ *bits).*

1. *The state-transition function h of SFMT is an $\mathbb{F}_2$-linear bijection, whose characteristic polynomial is factorized as*

$$\chi_h(t) = \chi_{19937}(t) \times \chi_{31}(t),$$

   where $\chi_{19937}(t)$ is a primitive polynomial of degree 19937 and $\chi_{31}(t)$ is a polynomial of degree 31.
2. *The state S is uniquely decomposed into a direct sum of h-invariant subspaces of degrees 19937 and 31*

$$S = V_{19937} + V_{31},$$

   where the characteristic polynomial of h restricted to V_{19937} is $\chi_{19937}(t)$.
3. *From any initial state s_0 not contained in V_{31}, the period P of the state transition is a multiple of the 24th Mersenne Prime $2^{19937} - 1$, namely $P = (2^{19937} - 1)q$ holds for some $1 \leq q \leq 2^{31} - 1$ (q may depend on s_0). The period of the output sequence is also P.*
   In this case, in addition, the output sequence of 128-bit integers of SFMT is at least 155-dimensionally equidistributed with defect q, in the sense of [6, §4.4].
4. *Let s_0 be the initial state of the SFMT, i.e., an array of 128-bit integers of length 156. If the 32 MSBs of the first 128-bit integer in s_0 is* `0x4d734e48`*, then $s_0 \notin V_{31}$ (cf. §2.2). In the initialization of SFMT, the corresponding 32 bits in s_0 is set to this (cf. §2.6).*
5. *$\chi_h(t)$ has 8928 nonzero terms (which is much larger than 135 in the case of MT19937), and $\chi_{19937}(t)$ has 9991 nonzero terms.*

3.1 Period

Proposition 2. *Any bit of the 8-bit integer stream generated by CryptMT Ver.3 has a period that is a multiple of* $2^{19937} - 1$.

Proof. Put $Q := 2^{19937} - 1$. Assume the converse, so there exists one bit among the 8 bits whose period is not a multiple of Q, which we call a short-period bit.

Let us denote by $h_0, h_1, h_2, \ldots$ the output 8-bit integer sequence of CryptMT Ver.3. If we consider CryptMT Ver.3 as a 64-bit integer generator (see §2.4), then its outputs $z_0, z_1, z_2, \ldots$ determine $h_0, h_1, h_2, \ldots$ by

$$\begin{aligned} z_0 &= (h_{13}, h_{12}, h_9, h_8, h_5, h_4, h_1, h_0) \\ z_1 &= (h_{15}, h_{14}, h_{11}, h_{10}, h_7, h_6, h_3, h_2) \\ z_2 &= (h_{29}, h_{28}, h_{25}, h_{24}, h_{21}, h_{20}, h_{17}, h_{16}) \\ z_3 &= (h_{31}, h_{30}, h_{27}, h_{26}, h_{23}, h_{22}, h_{19}, h_{18}) \\ &\vdots \end{aligned} \tag{5}$$

From this, we see that the bits in $z_0, z_2, z_4, \ldots$ that corresponds to the short-period bit (there are 8 bits for each) has a period not a multiple of Q (since it is obtained by taking every 16-th h's). This implies that each of the corresponding 8 bits in $z_0, z_1, z_2, z_3, \ldots$ have a period not a multiple of Q.

We use Theorem A.1 in [6] (or equivalently Theorem 1 in [9]) to show that any two bits among the 64 bits in z_i have a period that is a multiple of Q (as a 2-bit integer sequence), which proves this proposition. We consider CryptMT Ver.3 as a 64-bit integer stream generator. Then it satisfies the conditions in the theorem, with $n = 155$, $Q = 2^{19937} - 1$, $q < 2^{31}$, and $Y = \mathbb{F}_2{}^{128}$. If we define the mapping $g : Y \to B$ in Theorem A.1 by setting $B := \mathbb{F}_2{}^2$ and

$$g : \mathbf{y} \mapsto \text{any fixed two bits in } \mathrm{LSB}_{32}^{16}(\mathbf{y} \oplus (\mathbf{y} >>_{32} 16)),$$

then $r = 1/4$ and the inequality

$$r^{-156} = 2^{312} > q \times \#(Y)^2 \ (< 2^{31} \times 2^{256})$$

implies that any pair of bits in the 64 bits has period of a multiple of Q, by Theorem A.1.

3.2 Time-memory-trade-off attack

A naive time-memory-tradeoff attack consumes the computation time of roughly the square root of the size of the state space, which is $O(\sqrt{2^{19968+128}}) = O(2^{10048})$ for the Version 3.

3.3 Dimension of Equidistribution

Proposition 1 shows that SFMT satisfies all conditions in §4.2–§4.3 of [6], with period $P = (2^{19937} - 1)q$ $(1 \leq q < 2^{31})$ and $n = 155$-dimensional equidistribution with defect $d = q$. Proposition 4.4 in [6] implies that the output 64-bit integer sequence of CryptMT Version 3 is 156-dimensionally equidistributed with defect $q \cdot 2^{128} < 2^{159}$, and hence 1241-dimensionally equidistributed as 8-bit integers. (The argument here appears also in §2.1 of [9].)

3.4 Correlation attacks and distinguishing attacks

By Corollary 4.7 in [6], if we consider a simple distinguishing attack to CryptMT Ver.3 of order ≤ 155, then its security level is $2^{19937\times 2}$, since $P/d = 2^{19937} - 1$.

Because of the 156-dimensional equidistribution property, correlation attacks seem to be non-applicable. See §4.5 of [6] for more detail.

3.5 Algebraic degree of the filter

Proposition 4.11 in [6] is about the multiplicative filter, so it is not valid for CryptMT Ver.3 as it is. However, since the filter of the Version 3 introduces more bit-mixing than the original multiplicative filter, we guess that each bit of the output of CryptMT Ver.3 would have high algebraic degree, close to the upper bound coming from the number of variables. Algebraic attacks and Berlekamp-Massey attacks would be infeasible, by the same reasons stated in §4.9 and §4.10 of [6].

3.6 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, Dragon (these are the five candidates in eSTREAM software cipher phase 3 permitting 256-bit Key), SNOW2.0 [3] and AES (counter-mode), in three different CPUs: Intel Core 2 Duo, AMD-Athlon X2, and Motorola PowerPC G4, using eSTREAM timing-tool [2]. The data are listed in Table 1. Actually, they are copied from Bernstein's page [1]. The number of cycles in Key set-up and IV set-up are also listed.

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). Note that PowerPC is replaced with Intel CPUs in the present version of Mac PCs.

Table 1. Summary from eSTREAM benchmark by Bernstein[1]

Primitive	Core 2 Duo			AMD Athlon 64 X2			Motorola PowerPC G4		
	Stream	Key setup	IV setup	Stream	Key	IV	Stream	Key	IV
CryptMT3	2.95	61.41	514.42	4.73	107.00	505.64	9.23	90.71	732.80
HC-256	3.42	61.31	83805.33	4.26	105.11	88726.20	6.17	87.71	71392.00
SOSEMANUK	3.67	848.51	624.99	4.41	1183.69	474.13	6.17	1797.03	590.47
SNOW-2.0	4.03	90.42	469.02	4.86	110.70	567.00	7.06	107.81	719.38
Salsa20	7.12	19.71	14.62	7.64	61.22	51.09	4.24	69.81	42.12
Dragon	7.61	121.42	1241.67	8.11	120.21	1469.43	8.39	134.60	1567.54
AES-CTR	19.08	625.44	18.90	20.42	905.65	50.00	34.81	305.81	34.11

4 Conclusion

We modified the mother generator, the filter, and the initialization of CryptMT and CryptMT Ver.2 so that they fit to the parallelism of modern CPUs, such as single-instruction-multiple-data operations and pipeline processing.

The proposed CryptMT Ver.3 is 1.8 times faster than the first version (faster than SNOW2.0 on Core 2 Duo and AMD Athlon platform), while the astronomical period $\geq 2^{19937}-1$ and the 1241-dimensional equidistribution property (as a 8-bit integer generator) are guaranteed. The Key-size and the IV-size can flexibly chosen from 128 bits to 2048 bits for each. The size of the state and the length of the period makes time-memory-trade-off attacks infeasible, and the high non-linearity introduced by the integer multiplication would make the algebraic attacks and Berlekamp-Massey attacks impossible. CryptMT has no look-up tables, and hence has resistance to the cache-timing attacks.

A shortcoming of CryptMT Ver.3 might be in the size of consumed memory (nearly 2.6KB), but it does not matter in usual computers (of course it does matter in some applications, though).

5 Intellectual Property Status

CryptMT is patent-pending. Its property owners are Hiroshima University and Ochanomizu University. However, the inventors (i.e., the authors of this manuscript) had the following permission from the owners:

- CryptMT is free for non-commercial use.
- If CryptMT survives in the final portfolio of the stream ciphers in eSTREAM competition, then it is free even for commercial use.

The inventors' wish is that this algorithm be freely and widely used in the community, similarly to Mersenne Twister PRNG [4] invented by the first and the third authors.

References

1. Bernstein, D.J. http://cr.yp.to/streamciphers/timings.html.
2. eSTREAM – The ECRYPT Stream Cipher Project – Phase 3. http://www.ecrypt.eu.org/stream/index.html
3. Ekdahl, P., Johansson, T. A New Version of the Stream Cipher SNOW, Selected Areas in Cryptography, SAC 2002, Springer Verlag, LNCS 2595, pp. 47–61, 2002.
4. Matsumoto, M. and Nishimura, T. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation, 8 (1998) 3–30.
5. Matsumoto, M., Nishimura, T., Saito, M. and Hagita, M. Cryptographic Mersenne Twister and Fubuki stream/block cipher, http://eprint.iacr.org/2005/165. This is an extended version of "Mersenne Twister and Fubuki stream/block cipher" in http://www.ecrypt.eu.org/stream/cryptmtfubuki.html.
6. Matsumoto, M. Saito, M., Nishimura, T. and Hagita, M. Cryptanalysis of CryptMT: Effect of Huge Prime Period and Multiplicative Filter, SASC2006 Conference Volume http://www.ecrypt.eu.org/stream/cryptmtfubuki.html.
7. Matsumoto, M., Saito, M., Nishimura, T. and Hagita, M. CryptMT Version 2.0: a large state generator with faster initialization, SASC2006 Conference Volume http://www.ecrypt.eu.org/stream/cryptmtfubuki.html.
8. Matsumoto, M., Saito, M., Nishimura, T. and Hagita, M. CryptMT Stream Cipher Version 3, SASC2007 Conference Volume http://www.ecrypt.eu.org/stream/cryptmtp3.html.
9. Matsumoto, M., Saito, M., Nishimura, T. and Hagita, M. A Fast Stream Cipher with Huge State Space and Quasigroup Filter for Software, Proceedings of SAC2007, LNCS4876 (2007), 245–262.
10. Saito, M. and Matsumoto, M. SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator, to appear in Proceedings of MCQMC2006, Springer-Verlag.