

CRYPTMT STREAM CIPHER VERSION 3

MAKOTO MATSUMOTO, MUTSUO SAITO, TAKUJI NISHIMURA,
AND MARIKO HAGITA

ABSTRACT. In the previous manuscripts, we proposed CryptMT pseudorandom number generator (PRNG) for a stream cipher, which is a combination of (1) an \mathbb{F}_2 -linear generator of a wordsize-integer sequence with a huge state space, and (2) a filter with one wordsize memory, based on the accumulative integer multiplication and extracting some most significant bits from the memory. No valid attacks have been reported so far.

In this manuscript, we propose a modification of the algorithm to speed up the generation and the initialization, by (1) using the parallelisms such as pipelining and single-instruction-multiple-data (SIMD) operations included in modern CPUs, (2) in the filter, raising the ratio between the numbers of output bits and the input bits from 1/4 to 1/2.

As a result, we propose the CryptMT Version 3 which has 1.8 times faster generation and 48.7 times faster initialization than the Version 1, while the security level seems comparable to the original version. The generation/initialization speed is comparable to the fast stream ciphers such as SNOW2.0. Moreover, CryptMT Version 3 is proved to have a period that is a multiple of $2^{19937} - 1$ and strong resistance to the standard attacks.

1. INTRODUCTION

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

Our proposal [4][5] is to combine a huge state linear generator (called the mother generator) and a filter with memory, as shown in Figure 1.

Definition 1.1. (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 $x_0, x_1, x_2, \dots \in X$. 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 $f : Y \times X \rightarrow Y$ 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 i -th step is given by $g(y_i)$, where $g : Y \rightarrow O$ is the output function which converts the content of the memory to an output symbol in O .

Date: June 29, 2006.

Key words and phrases. Cryptographic Mersenne Twister, CryptMT, Simple Fast Mersenne Twister, stream cipher, fast initialization, booter.

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

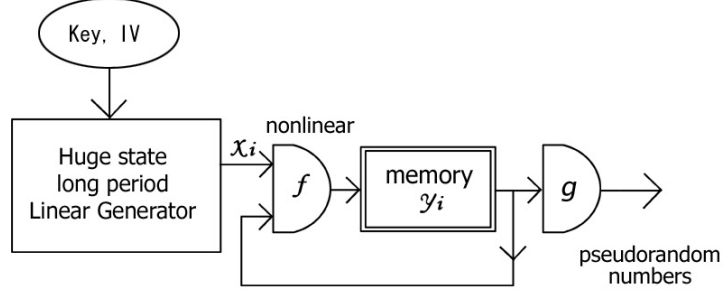


FIGURE 1. Combined generator = linear generator + filter with memory.

In the previous manuscripts, we chose the mother generator to be Mersenne Twister [3], which generates a sequence of 32-bit integers by an \mathbb{F}_2 -linear recursion. The filter is given by

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

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 .

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.

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 ,

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

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

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

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

In the following, we often use functions such as

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

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

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

is also called a perm-shift, where $i_3i_2i_1i_0$ 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 $\text{MSB}^n(x)$. Similar notation $\text{LSB}^n(x)$ is also used. For a 128-bit integer \mathbf{x} , we define

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

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

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

2.2. Simple Fast MT. The Version 3 adopts the following mother generator, named Simple Fast Mersenne Twister (SFMT) [2].

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

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

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 $(\mathbf{x}_{M+j}[3][2] \gg S)$ and $(\mathbf{x}_{M+j}[1][0] \gg S)$, as explained above. The third term is a rotation 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 $\text{MASK} = \text{ffdfafdf f5dabfff ffdbffff ef7bffff}$ in the hexa-decimal notation. Such a mask is necessary to break the symmetricity (i.e., without such asymmetricity, 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 $\mathbf{x}_0[3] = \text{0x4d734e48}$, 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.

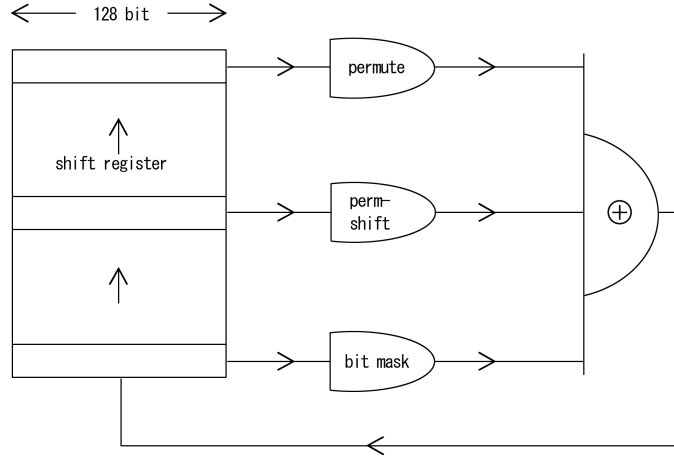


FIGURE 2. The mother generator: Simple Fast Mersenne Twister.

permute: $\mathbf{y} \mapsto \mathbf{y}[0][3][2][1]$.perm-shift: $\mathbf{y} \mapsto \mathbf{y}[2][0][3][1] \oplus (\mathbf{y} \gg_{64} 3)$.bit-mask: `ffdfafdf f5dabfff ffdbffff ef7bffff`

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

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

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

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

which is essentially the multiplication of 33-bit odd integers. Let S be the set of odd integers in $\mathbb{Z}/2^{33}$. We have a bijection

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

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: $X = Y = (\mathbb{Z}/2^{32})^4$, 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

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

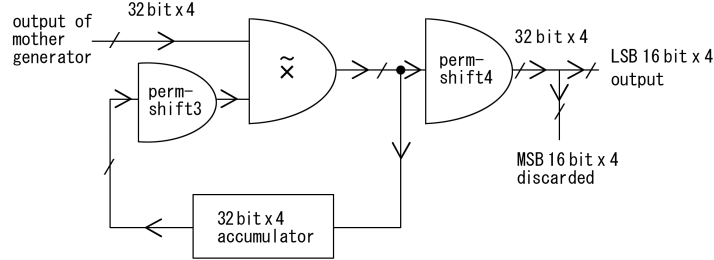


FIGURE 3. Filter of CryptMT Ver.3.

perm-shift3: $\mathbf{y} \mapsto \mathbf{y} \oplus (\mathbf{y}[0][3][2][1] \gg_{32} 1)$.

perm-shift4: $\mathbf{y} \mapsto \mathbf{y} \oplus (\mathbf{y} \gg_{32} 16)$.

$\tilde{\times}$: multiplication of 33-bit odd integers.

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

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

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

$$(4) \quad g(\mathbf{y}) := \text{LSB}_{32}^{16}(\mathbf{y} \oplus (\mathbf{y} \gg_{32} 16)).$$

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

$$\begin{aligned} \text{LOWER16} &:= (0x0000ffff, 0x0000ffff, 0x0000ffff, 0x0000ffff) \\ \text{UPPER16} &:= (0xffff0000, 0xffff0000, 0xffff0000, 0xffff0000) \end{aligned}$$

be the 128-bit masks.

Let $\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_{2i}, \mathbf{y}_{2i+1}, \dots$ 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

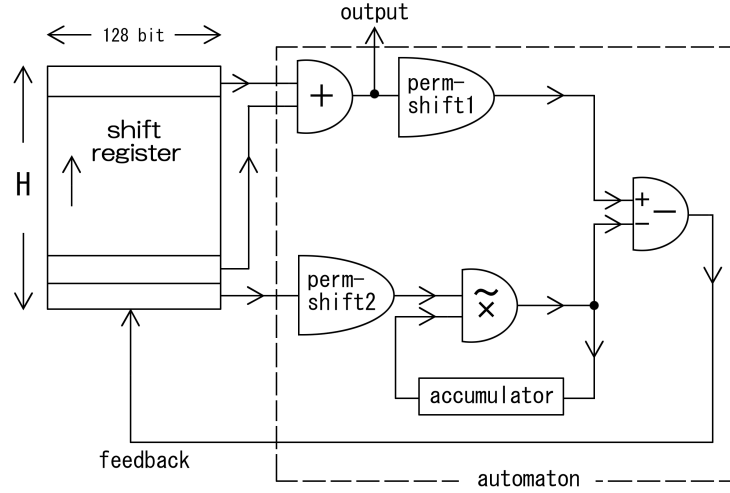


FIGURE 4. Booter of CryptMT Ver.3.

perm-shift1: $\mathbf{x} \mapsto (\mathbf{x}[2][1][0][3]) \oplus (\mathbf{x} \gg_{32} 13)$.

perm-shift2: $\mathbf{x} \mapsto (\mathbf{x}[1][0][2][3]) \oplus (\mathbf{x} \gg_{32} 11)$.

$\tilde{\times}$: multiplication of 33-bit odd integers.

generate the i -th 128-integer output \mathbf{z}_i , by the formula

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

where \mid 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 [6] 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 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 $\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{H-1}$ 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

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

where

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

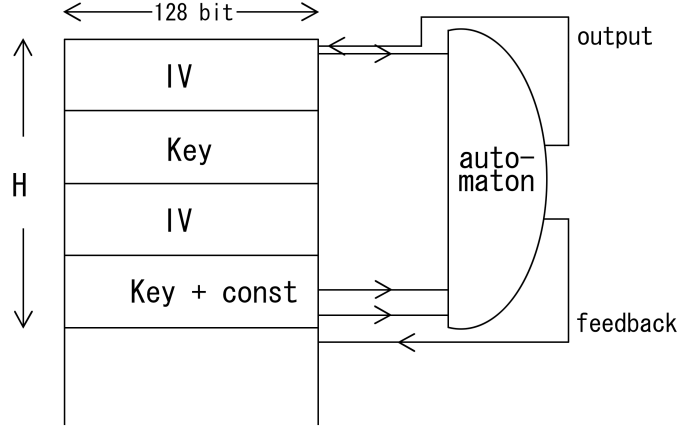


FIGURE 5. Key and IV set-up. The IV-array and Key-array are concatenated and copied to an array twice. Then, a constant is added to the bottom of the second copy of the key to break a possible symmetry.

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 same with the Key-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, 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 $2 \times (\text{IV-size} + \text{Key-size (in bits)})/128$, 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.

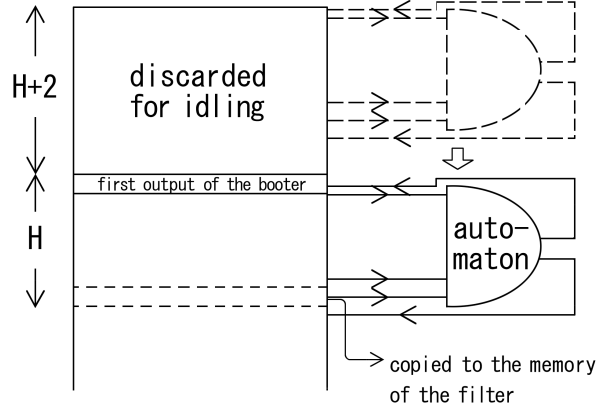


FIGURE 6. After the Key and IV set-up.

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.

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) of SFMT with $\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_{N-1}$ being the array of length N indicated in Figure 7, and produces $\mathbf{x}_N, \mathbf{x}_{N+1}, \dots$. 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 $\mathbf{x}_{N+1}, \mathbf{x}_{N+2}, \dots$ to the filter.

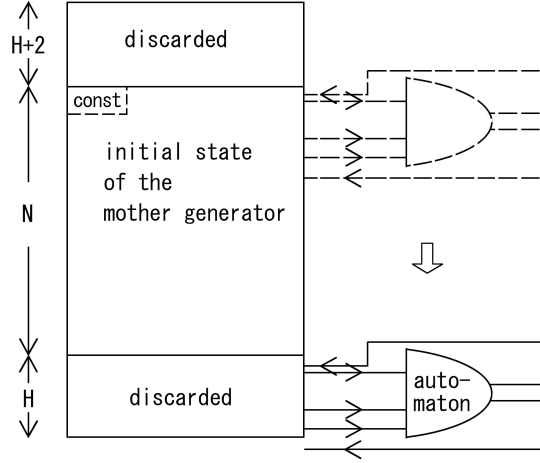


FIGURE 7. Initialization of the SFMT mother generator.

3. RESISTANCE OF CRYPTMT VER.3 TO STANDARD ATTACKS

The cryptanalysis developed in §4 in [5] for CryptMT is also valid for the Version 3. We list some properties of the SFMT (§2.2) required in the following cryptanalysis. Details of proofs should be written elsewhere.

Proposition 3.1. 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 155-dimensionally equidistributed with defect q , in the sense of [5, §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 3.2. 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 8 possible bits) of the 8-bit integer stream whose period is not a multiple of Q . Let us denote by h_0, h_1, h_2, \dots the output 8-bit integer sequence of CryptMT Ver.3.

If we consider CryptMT Ver.3 as a 64-bit integer generator, then its outputs z_0, z_1, z_2, \dots determines h_0, h_1, h_2, \dots by

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

From this, we see that the corresponding bits in z_0, z_2, z_4, \dots (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 $z_0, z_1, z_2, z_3, \dots$ have the period not a multiple of Q .

We use Theorem A.1 in [5] 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 in $g : Y \rightarrow B$ in Theorem A.1 by setting $B := \mathbb{F}_2^2$ and

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

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 at least Q , by Theorem 4.1. loc. cit. \square

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 3.1 shows that SFMT satisfies all conditions in §4.2–§4.3 of loc. cit., with period $P = (2^{19937} - 1)q$ ($1 \leq q < 2^{31}$) and $n = 155$ -dimensional equidistribution with defect $d = q$. Proposition 4.4 (loc. cit.) implies that the output 64-bit integer sequence of CryptMT Version 3 (more precisely, its indistinguishable modification stated in Assumption 4.3 there) is 156-dimensionally equidistributed with defect $q \cdot 2^{128} < 2^{159}$, and hence 1248-dimensionally equidistributed as 8-bit integers.

3.4. Correlation attacks and distinguishing attacks. By Corollary 4.7 (loc. cit.), 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 seems to be non-applicable. See §4.5 loc. cit. for more detail.

3.5. Algebraic degree of the filter. Proposition 4.11 (loc. cit.) 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 loc. cit.

4. PERFORMANCE COMPARISON

We used the performance testing tool from eSTREAM [1] to compare the speed of Focused Phase 2 candidate algorithms with CryptMT Ver.3, in the two different platforms, i.e., Pentium-M 1.4GHz CPU and AMD Athlon 64 CPU, with the compiler gcc version 3.4.4.

Table 1 lists the results for Pentium-M. The generation speed of CryptMT Ver.3 (with the primitive name CryptMT3) is about 1.8 times faster than the first version, and the IV-setup is about 48.7 times faster. This table shows that the Version 3 is comparable to other fast generators in Pentium-M.

Table 2 lists the results for AMD Athlon 64, which seems to have faster SIMD instructions than Pentium-M. As a result, CryptMT Ver.3 is the fourth fastest generator among the compared algorithms in this platform.

5. 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 AMD Athlon on platforms with fast SIMD instructions), while the astronomical period $\geq 2^{19937} - 1$ and the 1248-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 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 short-coming 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).

Primitive	Key	IV	Stream	40 bytes	576 bytes	1500 bytes	Key setup	IV setup
Py	128	64	2.63	138.62	12.03	6.22	2360.36	4241.39
Py	256	128	2.63	141.98	12.26	6.31	2586.97	4374.11
Py6	128	64	2.69	72.73	7.27	4.45	805.75	1908.94
Py6	256	128	2.69	79.8	7.76	4.64	969.88	2184.92
HC-256	128	128	4.44	1680.85	120.49	49.18	48.15	66724.6
HC-256	256	128	4.44	1680.4	120.59	49.16	44.01	66700
SNOW-2.0	128	128	4.68	28.23	5.97	5.3	69.52	710.39
SNOW-2.0	256	128	4.68	28.83	6.01	5.32	100.72	754.33
SOSEMANUK	128	64	5.09	35.68	10.13	8.61	1333.08	807.34
SOSEMANUK	256	128	5.09	33.12	9.95	8.54	1257.13	701.87
Salsa20/8	128	64	5.5	19.14	5.67	5.81	47.75	34.31
Salsa20/8	256	64	5.5	19.14	5.67	5.81	46.54	34.31
CryptMT3	256	128	5.78	29.25	10.53	10.3	45.81	637.13
CryptMT3	128	128	5.79	26.71	10.34	10.21	40.22	532.12
Phelix	128	128	6.63	25.43	8	7.13	398.39	875.61
Phelix	256	128	6.63	25.43	8	7.13	396.97	875.61
Salsa20/12	128	64	7.57	22.24	7.72	7.94	45.86	33.7
Salsa20/12	256	64	7.57	22.24	7.72	7.94	45.86	33.7
CryptMT	128	128	10.48	798.8	82.16	34.24	35.18	31048.55
CryptMT	256	128	10.48	798.8	82.16	34.24	44.01	31048.55
CryptMT-v2	128	128	10.7	73.8	23.16	17.65	22487.87	2145.47
CryptMT-v2	256	128	10.7	73.8	23.16	17.64	22578.87	2145.38
Salsa20	128	64	11.73	29.16	11.93	12.22	45.73	35.82
Salsa20	256	64	11.73	29.14	11.93	12.22	47.69	35.82
LEX	128	128	11.87	24.33	13.48	12.49	309.94	462.38
Dragon	128	128	12.67	81.73	30.84	28.48	191.04	2158.37
Dragon	256	128	12.67	82.53	30.83	28.48	189.24	2170.61

TABLE 1. Comparison of performance: cycles per byte, measured by ECRYPT tools on Intel(R) Pentium(R)-M processor, 1.4GHz.

6. 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 is selected as one of the recommendable stream ciphers by eSTREAM, then it is free even for commercial use.

In the Phase-1 selection, eSTREAM stated: “While we do not rule out submissions covered by IP, our preference is that the very limited cryptanalytic time available should be focused on ciphers that are guaranteed to be unencumbered. Thus, we are unable to recommend algorithms covered by IP as focus ciphers” (see <http://www.ecrypt.eu.org/stream/endpointofphase1.html>).

Here we the inventors, with approval by the owners of the IP, guarantee that CryptMT (including Versions 1, 2, and 3) will be free, if eSTREAM recommends it (say, even as a special purpose generator). Thus, we would like to ask eSTREAM committee to consider CryptMT as “unencumbered” or “not covered by IP.”

Primitive	Key	IV	Stream	40 bytes	576 bytes	1500 bytes	Key setup	IV setup
Salsa20/8	128	64	3.83	12.11	3.94	4.03	28.93	12.9
Salsa20/8	256	64	3.83	12.11	3.94	4.03	27.9	12.9
SOSEMANUK	128	64	4.39	24.99	8.55	7.53	1160.78	492.86
SOSEMANUK	256	128	4.39	23.33	8.26	7.3	1156.77	440.17
HC-256	128	128	4.51	2176.03	155.28	63.13	41.2	87171.5
HC-256	256	128	4.51	2189.56	156.27	63.15	33.91	87237.5
CryptMT3	128	128	4.73	19.8	8.49	7.67	67.71	392.14
CryptMT3	256	128	4.73	22.49	8.68	7.74	76.71	499.06
SNOW-2.0	128	128	4.89	23	5.86	5.4	43.8	516.42
SNOW-2.0	256	128	4.89	23.49	5.89	5.41	66.91	535.6
Phelix	128	128	5.66	31.73	9.63	8.63	295.12	771.84
Phelix	256	128	5.66	30.73	9.57	8.6	292.81	734.63
Py	128	64	6.16	157.07	16.57	10.12	2748.28	5263.93
Py	256	128	6.16	162.33	16.94	10.26	3054.31	5473.05
Py6	128	64	6.17	53.47	9.31	7.34	855.43	1522.29
Salsa20/12	128	64	6.17	20.83	6.31	6.58	28.23	12.96
Py6	256	128	6.17	57.73	9.6	7.45	1143.9	1692.59
Salsa20/12	256	64	6.17	20.83	6.31	6.58	28.41	12.9
Dragon	128	128	8.27	64.68	27.97	26.43	180.03	1563.09
Dragon	256	128	8.27	64.6	27.97	26.43	179.62	1559.97
LEX	128	128	8.44	17.2	9.58	8.87	190.22	330.44
Salsa20	128	64	9.78	26.29	9.86	10.21	28.23	12.84
Salsa20	256	64	9.78	26.44	9.92	10.21	28.43	12.84
CryptMT-v2	128	128	11.06	82.83	19.72	14.68	20815.92	2672.08
CryptMT-v2	256	128	11.06	82.83	19.72	14.68	20877.08	2672.08
CryptMT	128	128	11.44	626.81	56.51	28.16	26.2	24633.22
CryptMT	256	128	11.44	626.81	56.5	28.17	38.62	24633.22

TABLE 2. Comparison of performance: cycles per byte, measured on AMD Athlon(tm) 64 Processor 3400+.

The inventors' wish is that this algorithm be freely and widely used in the community, similarly to Mersenne Twister PRNG [3] invented by the first and the third authors.

REFERENCES

- [1] eSTREAM Optimized Code Howto, <http://www.ecrypt.eu.org/stream/perf/>.
- [2] Saito, M. and Matsumoto, M. Simple and Fast Mersenne Twister. To be presented at Monte Carlo and Quasi Monte Carlo Method 2006.
- [3] 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.
- [4] 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” submitted for eSTREAM proposal <http://www.ecrypt.eu.org/stream/>.
- [5] 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/>.
- [6] 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/>.

14 MAKOTO MATSUMOTO, MUTSUO SAITO, TAKUJI NISHIMURA, AND MARIKO HAGITA

DEPARTMENT OF MATHEMATICS, HIROSHIMA UNIVERSITY, HIROSHIMA 739-8526, JAPAN
E-mail address: `m-mat@math.sci.hiroshima-u.ac.jp`

DEPARTMENT OF MATHEMATICS, HIROSHIMA UNIVERSITY, HIROSHIMA 739-8526, JAPAN
E-mail address: `saito@math.sci.hiroshima-u.ac.jp`

DEPARTMENT OF MATHEMATICS, YAMAGATA UNIVERSITY, YAMAGATA JAPAN
E-mail address: `nisimura@sci.kj.yamagata-u.ac.jp`

DEPARTMENT OF INFORMATION SCIENCE, OCHANOMIZU UNIVERSITY, TOKYO JAPAN
E-mail address: `hagita@is.ocha.ac.jp`