CRYPTOGRAPHIC MERSENNE TWISTER AND FUBUKI STREAM/BLOCK CIPHER

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

Abstract. We propose two stream ciphers based on a non-secure pseudorandom number generator (called the mother generator). The mother generator is here chosen to be the Mersenne Twister (MT), a widely used 32-bit integer generator having 19937 bits of internal state and period $2^{19937} - 1$.

One proposal is CryptMT, which computes the accumulative product of the output of MT, and use the most significant 8 bits as a secure random numbers. Its period is proved to be $2^{19937} - 1$, and it is 1.5-2.0 times faster than the most optimized AES in counter-mode.

The other proposal, named Fubuki, is designed to be usable also as a block cipher. It prepares nine different kinds of encryption functions (bijections from blocks to blocks), each of which takes a parameter. Fubuki encrypts a sequence of blocks (= a plain message) by applying these encryption functions iteratedly to each of the blocks. Both the combination of the functions and their parameters are pseudorandomly chosen by using its mother generator MT. The key and the initial value are passed to the initialization scheme of MT.

1. Introduction

In this paper, we consider cryptographic systems implemented in software. We assume a 32-bit CPU machine with fast multiplication of words, and a moderate size of working area (about 4K bytes).

In a narrow sense, a stream cipher system is to generate cryptographically secure pseudorandom numbers (PN) from a shared key, and take exclusive-or with the plain message to obtain ciphered message. One way to generate such PN is to use a non-secure generator like LFSR (which we call the mother generator), initialize it by using the key, and then filter its outputs, i.e., apply some complicated functions to obtain a secure sequence.

Along this line, we propose to use a GF(2)-linear generator whose internal state consists of 19937 bits, Mersenne Twister (MT) (see §3 for the detail). MT is invented by two of the authors [4]. It has period $2^{19937} - 1$ and uniform equidistribution property upto 623 dimension. Its initialization scheme is improved to accept an array of any length as an initial seed in 2002. MT is widely accepted in

Date: June 1, 2005.

Key words and phrases. Mersenne Twister, non-secure random number generator, stream cipher, CryptMT, Fubuki, AES.

The cipher systems CryptMT and Fubuki in this paper are proposed to ECRYPT Stream Cipher Proposal `http://www.ecrypt.eu.org/stream/`. The reference codes and a prototype of this paper are available there. The first author was supported in part by JSPS Grant-In-Aid #14654021 and #16204002, and Hiroshima University President's Discretion Fund '05.

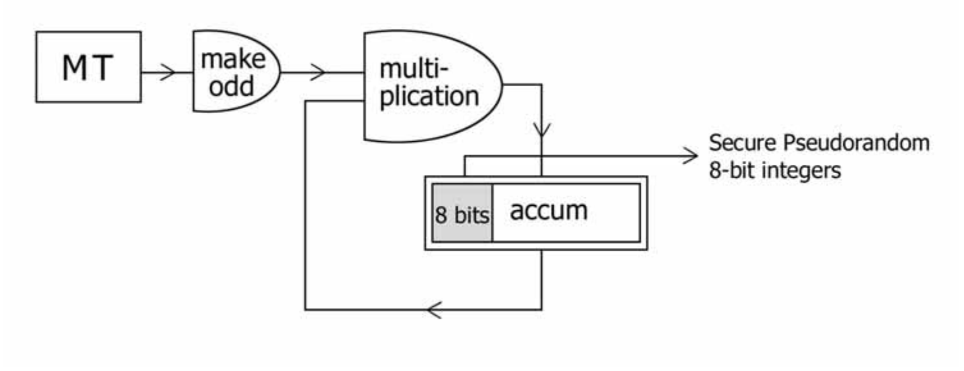

FIGURE 1. CryptMT: period $2^{19937}-1$, and twice faster than the optimized AES (Pentium-M, gcc -O3)

the society of MonteCarlo simulations, and implementations in C and many other languages are available from the homepage [5].

As described in this homepage, a way to generate a cryptographically secure PN sequence is to use MT, and compress its outputs by using, say, MD5 or SHA1.

2. CryptMT

The first proposal in this paper is even simpler. MT generates a sequence of unsigned 32 bit integers (which from now on we shall call words). The given key and initial value are concatenated and passed to the initialization scheme (§3) of MT. We prepare a variable `accum` of word size, which is set to 1 at the beginning (this may be any odd integer).

Then, we iterate the following process to obtain (probably) a cryptographically secure PN sequence of 8-bit integers (= byte):

(1) Generate one pseudorandom word `gen_rand` by MT.
(2) Multiply it to `accum`:

$$\texttt{accum} \leftarrow \texttt{accum} \times (\texttt{gen\_rand} \mid \texttt{1}).$$

(3) Output the most significant 8 bits of `accum`. Go to Step 1.

To raise the security, the first 64 bytes of the outputs are discarded.

Here the C-language-like notation "|" denotes bitwise-OR operation. This operation is to make the multiplier odd (otherwise, after several iterations, accum would be zero). Multiplication is considered modulo 2^{32}.

This method generates a PN sequence of bytes, which fits to the usual requirements for a stream cipher. We call this stream cipher CryptMT, meaning Cryptographic Mersenne Twister.

Our experiment shows that CryptMT is faster by a factor of 1.5–2.0 than the well-optimized counter-mode AES (see §6.1), widely known as `rijndael-alg-fst.c`. The size of the internal state of MT seems to be enough to make any kind of time-memory-trade-off attacks infeasible.

If all bits of `accum` were used (differently from the 8 bits as in CryptMT) then the sequence would not be cryptographically secure, since from the change of the `accum` we could recover the output of MT (except for the least significant bit), then by linear algebra one can decide the internal state after observing 19937 bits of the output. However, if only the most significant 8 bits after multiplication are observed, then we can not imagine how to obtain the internal state of MT.

It is important to use the most significant bits: the least significant bit is always 1, and the second bit of `accum` coincides with the summation (modulo 2) of the second bit of the outputs of MT so far, from which one could compute MT's internal state. The most significant bits seem to be safe, since the bit-diffusion pattern of the multiplication is from right to left, and most significant bits gather information of all the less significant bits of the two operands: `accum` and the output of MT.

The above gave a complete description of CryptMT, except for the description of the mother generator MT (§3). Security of CryptMT is largely depending on the mother generator MT and its initialization. The facts that (1) the size of internal state of MT is huge, (2) 3/4 of the output bits of MT are discarded, (3) MSBs after multiplication gather information of all bits, and (4) initialization is highly nonlinear, seem to imply high security, but we need more detailed study. The period of each of 8 bits of the output of CryptMT is $2^{19937}-1$ (see Appendix A).

Design rationales of CryptMT are

(1) use a fast linear generator, which has a huge state (e.g. thousands of bits), and
(2) filter its output by a finite state non-linear automaton which has a relatively small state (e.g. one word),
(3) Only a small fraction of the information of the state is output (e.g. 8 bits among 32 bits).

as seen in Figure 1. The former ensures the long period, and the latter ensures complicated bit-diffusion, under a compromise with the speed in software implementation.

3. Mersenne Twister

MT generates a PN word sequence by the GF(2)-linear recursion

$$w_{624+i} = w_{397+i} \oplus ((w_i \& \texttt{0x80000000}) | (w_{1+i} \& \texttt{0x7fffffff}))A \quad (i = 0, 1, 2, \ldots).$$

Here w_i ($i = 0, 1, 2, \ldots$) are 32-bit integers, each of which is considered as a 32-dimensional row vector over the two element field GF(2). The binary operator $\oplus$ denotes bitwise exclusive-or, i.e., addition as a vector.

The C-like hexadecimal notation 0x80000000 denotes the vector whose components are all zero except for the left most 1. Thus, $((w_i \& \texttt{0x80000000}) | (w_{1+i} \& \texttt{0x7fffffff}))$ is the row vector obtained by concatenating the MSB of w_i and all bits but the MSB of w_{1+i}. To this vector a constant 32×32 matrix A is multiplied from the right. This matrix A is of the form

$$\begin{pmatrix} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ a_{31} & a_{30} & \cdots & \cdots & a_0 \end{pmatrix},$$

and so the multiplication is computed by

$$\mathbf{x}A = \begin{cases} \text{shiftright}(\mathbf{x}) & \text{(if the LSB of } \mathbf{x} \text{ is 0)} \\ \text{shiftright}(\mathbf{x}) \oplus \mathbf{a} & \text{(if the LSB of } \mathbf{x} \text{ is 1),} \end{cases}$$

where $\mathbf{a}$ is a constant vector

$$\mathbf{a} = (a_{31}, a_{30}, \ldots, a_0) = \texttt{0x9908B0DF} \text{ in the hexadecimal notation.}$$

These constants are chosen so that the period of the sequence is $2^{19937} - 1$. The number of nonzero terms in the characteristic polynomial of the state transition function is 153.

Figure 2 illustrates the state transition of MT. The state consists of 623 words + 1 bit. The next state is obtained by shifting one words to the above, and insert a new word linearly computed from the discarded part and a middle term. In a software implementation, instead of shifting, we use round robin technique (i.e. to use a pointer) for the efficiency of the generation.

Advantages of this configuration over usual LFSR with coefficients, say, in GF(2^{32}), are (1) No need of costive operations like "multiplication modulo polynomials," (2) There is a fast algorithm to check the maximality of the period by paring and Galois theory [4], (3) The generation speed is independent of $\mathbf{a}$, which makes the parameter-search easier. This is different from LFSR, where the generation speed depends on a particular choice of the coefficients.

Remark 3.1. In the original MT, the output is transformed by a linear transformation to attain nearly optimally high-dimensional equidistribution at MSBs. This is called Tempering. We removed this transformation, since we think it non-necessary because of the application of complicated functions to the outputs of MT in the encryption process.

For initialization, we need to specify $w_0, w_1, \ldots, w_{623}$ as the initial state (to be precise, all the 31 bits of w_0 but the most significant bit are neglected in generating the next word, so the state space has $624 \times 32 - 31 = 19937$ bits). The output sequence of MT is $w_{624}, w_{625}, \ldots$, i.e., MT skips the contents of the initial state.

The initialization is particularly important for MT. Because of the linearity and the sparseness of the recursion of MT, if the initial state has too many zeroes, then the output sequence has same tendency for more than 10000 outputs. The 2002 version of MT [5] has an initialization function `init_by_array(u32 init_key[], int key_length)`, whose first argument is an array of 32-bit words with length given by the second argument, which is described as follows.

First, $w_0, \ldots, w_{623}$ are set to a fixed nontrivial value by a recursion

$$\begin{array}{lcl} w_0 & \leftarrow & 19650218, \\ w_i & \leftarrow & ((w_{i-1} \oplus (w_{i-1} >> 30)) + i) \times 1812433253 \end{array} \tag{1}$$

($1 \le i \le 623$), where w_i are considered as 32-bit unsigned integer variables, and every arithmetic operation is modulo 2^{32}. The notation $>> 30$ means the shift to the right by 30 bits. (The constant 19650218 is the birthday of one of the authors, chosen without reason. The other constant 1812433253 is a multiplier for a linear congruential generator [3, P.106], here chosen without reason.)

This recursion is chosen to have a good bit-diffusion property. The multiplication with constant has a good bit-mixing property, except for that the diffusion of the

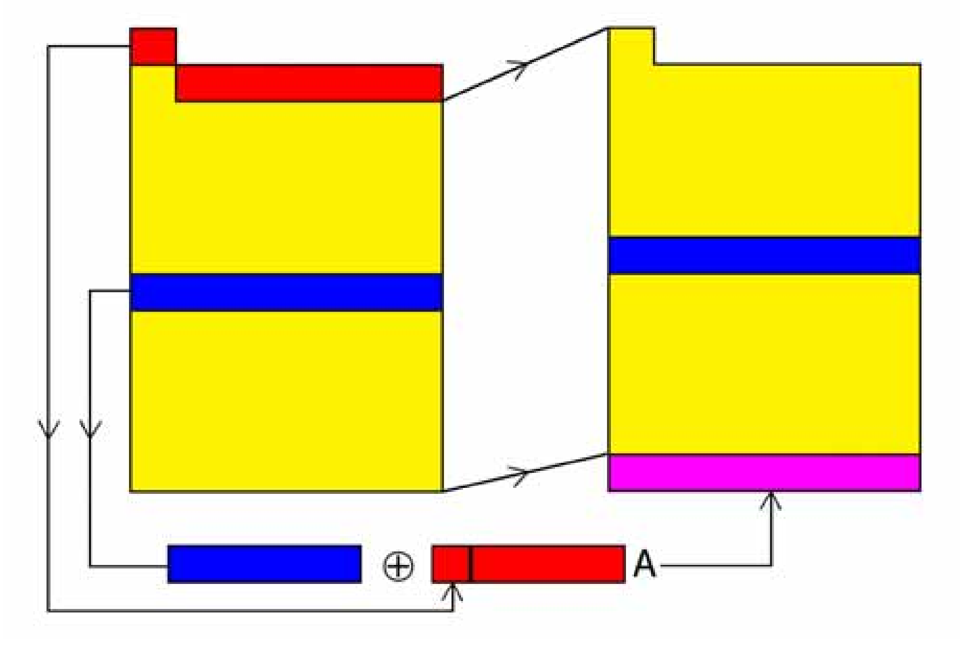

FIGURE 2. State transition of Mersenne Twister

bit information is always from right to left. The most significant two bits (which gather the information of all bits after multiplication) are sent to the least significant two bits of w_i by exclusive-or, to complement the multiplication. The assignment $w_{i-1} \mapsto w_i$ is bijective.

Addition with i is to avoid the following phenomenon. Suppose that i is not added in the recursion (1). Suppose that an initial value w_0 is chosen (although it is fixed to 19650218 in the above implementation) and let $w_0, w_1, \ldots, w_{623}$ be the generated sequence. Suppose that in another initialization another initial value w'_0 is chosen, which generates a sequence $w'_0, w'_1, \ldots, w'_{623}$. What we worry is that it may happen that $w'_0 = w_1$ by accident (or $w'_0 = w_2$, or alike), and then, $w'_i = w_{i+1}$ for $i = 0, 1, \ldots, 622$. Such similarity of the initial states yields correlated outputs for 10000 outputs or so according to the experiments. The addition with i avoids these phenomena.

The initial seed is given as an array `init_key[length]` of an arbitrary length `length` upto 64. The initialization scheme `init_by_array` rewrites the above $w_1, \ldots, w_{623}$ by the following recursive substitution:

$$\begin{aligned} w_i \quad \leftarrow \quad & (w_i \oplus (((w_{i-1} \oplus (w_{i-1} >> 30)) \times 1664525)) \\ & + \texttt{init\_key}[i \bmod \texttt{length}] + (i \bmod \texttt{length}) \end{aligned} \tag{2}$$

for $i = 1, 2, \ldots, 623$. Note that every multiplication or addition is done modulo 2^{32}. This recursion is chosen in the same spirit as (1), with adding `init_key[]` meanwhile. The reason why taking "i modulo `length`" at the last of the recursion is as follows. Suppose that we don't take modulo `length`. Suppose that one initialization is given by an array `init_key[]`, and another initialization is given another array of the twice length with the content being the two repetitions of the original array. Then the two initializations yield the same state. Such phenomenon is avoided by taking modulo `length`.

Then, we substitute the first word

$$w_0 \leftarrow w_{623},$$

and again apply a similar recursive substitution

$$w_i \leftarrow (w_i \oplus (((w_{i-1} \oplus (w_{i-1} >> 30)) \times 1566083941)) - i \tag{3}$$

for $i = 1, 2, 3, \ldots, 623$.

Finally, the most significant bit of w_0 is set to one, to avoid the zero initial state.

According to the experiments, this initialization has a good bit-distribution property. Any one bit of the change in the initial seed array `init_key[]` dramatically changes the initial state. The worst related keys seem to be those having difference only at the last word of `init_key[]`. However, the output of MT starts from w_{624}, which depends on w_{397}, which seem to be difficult to control because of at least $397 - (64 + 64)$ times application of (2) at the last word of `init_key[]`. (This $64 + 64$ is because the size of the key and the size of the initial value are upto 64 words). In addition, each word of the internal state is transformed by the nonlinear bit-mixing recurrence (3) on the key. It seems very difficult to utilize the technique of differential cryptanalysis with respect to the key.

Remark 3.2. The above initialization scheme was incorporated in 2002, and it actually admits an array of arbitrary length as the initial seeding vector [5]. This feature is to answer to the requests of financial engineers, who want to use the ascii code of the name of each companies in the stock markets, as the initial-seed array.

This initialization is not designed for cryptographic purpose, but it seems to have enough resistance, so we keep it as is. However, from the viewpoint of efficiency, this is redundant. A quicker initialization is possible. For example, we do not need the first round (setting constants to the state array) in the initialization.

4. Fubuki

The other proposal in this paper is Fubuki cipher system. The basic idea is: "software is soft, so we can make the choice of encryption functions as flexible as possible." This may be contrasted to Rijndael block cipher, where in each round the encryption function is fixed: just the key (to be exor-ed) is changed. The proposal of Fubuki is to change the parameters of each functions.

To fix the situation, we assume that a block consists of 4 words (i.e. 4 of 32-bit integers), but the reference implementation of Fubuki allows 4, 8 or 16 words as one block. We use the notations

$$W := \text{the set of 32 bit words}, \quad \mathcal{B} := W^4 = \text{the set of blocks}.$$

A plain message is a finite sequence of blocks, i.e. an element of $\mathcal{B}^L$, where $L \in \mathbb{N}$ is the length of the message.

This paper considers a stream cipher in a more general sense than taking exclusive-or with PN. Let $\mathcal{K}$ be the key space. In a typical case,

$$\mathcal{K} = W^4 = \text{the set of 128 bits}.$$

Definition 4.1. A stream cipher is a sequence of functions called encoding functions

$$E_i : \mathcal{B} \to \mathcal{B}, \quad i = 1, 2, \ldots,$$

and another sequence of functions called decoding functions

$$D_i : \mathcal{B} \to \mathcal{B}, \quad i = 1, 2, \ldots$$

such that $D_i \circ E_i = \mathrm{i}d$ (the identity function).

A plain message $B_1, B_2, \ldots$ is encoded into $E_1(B_1), E_2(B_2), \ldots$, and then decoded by $D_1(E_1(B_1)), D_2(E_2(B_2)), \ldots$. Each of E_i, D_i depends on both the key K and i.

Remark 4.2. If E_i and D_i are identical for any i, then the system is called a block cipher.

Remark 4.3. A stream cipher in Definition 4.1 can be used to generate a (possibly cryptographically secure) PN, by merely encoding a message consisting of, say, all zero.

The basic strategy of Fubuki is to compose several simple encryption functions, i.e., Shannon's "product" in his 1949 paper.

We use the following definition, which is nothing but a usual block cipher system (if we consider the parameter set as the set of keys).

Definition 4.4. A primitive encryption family PF with a parameter set $\mathcal{P}$ is a mapping

$$PF : \mathcal{P} \times \mathcal{B} \to \mathcal{B}$$

with its inverse family PF'

$$PF' : \mathcal{P} \times \mathcal{B} \to \mathcal{B}$$

such that for all $P \in \mathcal{P}$ and $B \in \mathcal{B}$

$$PF'(P, (PF(P, B))) = B$$

holds.

The size of $\mathcal{P}$ may vary among different PEFs.

Let us denote by

$$PF(P, -) : \mathcal{B} \to \mathcal{B}, \quad B \mapsto PF(P, B)$$

the bijection associated to the PF with parameter P.

Fubuki prepares nine different primitive encryption families. Four of them are designed to diffuse the bit-information mainly inside each word in the block (word-wise PEF), four of them are designed to mix the information of words (inter-word PEF), and the last one is designed to cut off the incidence relation of bits in each word (vertical rotate, denoted by $PF^{\text{v-rot}}$).

The given key and the given initial value are passed to the initialization of the mother generator MT. Using the non-secure PN sequence generated by MT, Fubuki pseudorandomly selects one of the four wordwise PEFs, say PF_1^{word}, and its parameter P_1^{word}, and apply $PF_1^{\text{word}}(P_1^{\text{word}}, -)$ to B_1. Then similarly select one of the four inter-word PEFs, say PF_1^{inter}, and its parameter P_1^{inter}, and apply it to the result. Then apply $PF^{\text{v-rot}}$ with pseudorandomly selected parameter $P_1^{\text{v-rot}}$.

This is one round of Fubuki encryption, and it is repeated several times. The choice in the reference code is four times iteration for each block. Thus, the block B_i is encoded by applying

$$E_i := \mathrm{Round}_{4,i} \circ \mathrm{Round}_{3,i} \circ \mathrm{Round}_{2,i} \circ \mathrm{Round}_{1,i}$$

where each round is given by

$$\mathrm{Round}_{j,i} = PF^{\text{v-rot}}(P^{\text{v-rot}}_{j,i}, -) \circ PF^{\text{inter}}_{j,i}(P^{\text{inter}}_{j,i}, -) \circ PF^{\text{word}}_{j,i}(P^{\text{word}}_{j,i}, -) \quad (j = 1, 2, 3, 4),$$

where PEFs and their parameters are pseudorandomly chosen by MT. (The $PF^{\text{inter}}_{j,i}$ is uniformly pseudorandomly chosen from four inter-word PEFs and its parameter is uniformly chosen from its parameter space. Similarly for $PF^{\text{word}}_{j,i}$.)

The decoding function D_i is its inverse, given by

$$E_i := \mathrm{Round}'_{1,i} \circ \mathrm{Round}'_{2,i} \circ \mathrm{Round}'_{3,i} \circ \mathrm{Round}'_{4,i}$$

where each round is given by

$$\mathrm{Round}'_{j,i} = PF^{\text{word}'}_{j,i}(P^{\text{word}}_{j,i}, -) \circ PF^{\text{inter}'}_{j,i}(P^{\text{inter}}_{j,i}, -) \circ PF^{\text{v-rot}'}(P^{\text{v-rot}}_{j,i}, -) \quad (j = 1, 2, 3, 4),$$

where $'$ denotes inverse PEFs.

The design rationale of Fubuki is as follows.

(1) An idea is to choose simplest operations (i.e. those in the instruction set of typical CPU, such as exor or multiplication) as the building blocks of PEF.

Any complicated operation is made from simple operations, so freely composing simple ones seems better than fixing one way.

(2) However, if it is too free, then (with very small probability) it may happen that one same PEF is selected all the time. So, there should be a trade-off between freedom to choose a combination of PEFs and restriction to assure good bit-information diffusion.

Fubuki did this by making each PEF a combination of a few simple operations.

(3) Fubuki has no S-boxes. In some sense, the integer multiplication replaces the S-boxes. The integer multiplication has a good and fast bit-diffusion property. It has two weakness: (1) the bit-diffusion is only from the right bits to the left ones, (2) it is (bi)-linear, so differential cryptanalysis is valid. However, Fubuki compensates these by (1) suitable bit-operations with left-to-right diffusion property and (2) combining exclusive-or to make it non-linear.

A recent study warns that any cryptographic system in a fast implementation using a large lookup table for S-box is vulnerable by cache-timing attack [6] [2]. This method breaks AES. (Fubuki has two tables of 32 words, which may leak some information. We need further study).

(4) Fubuki consumes far (e.g. 13 times) more PNs (from its mother generator) than the size of the plain message: since the parameter space of each PEF is large (actually we arranged the size to be nearly the same with one block), every round consumes three times block-size of PNs.

This redundancy makes it difficult to guess the internal state of the mother generator, even by chosen-plain text attack with chosen initial values.

Actually, Fubuki has an aspect of block cipher. Suppose that the key and the initial value are fixed, and these are repeatedly used in a stream cipher for different texts (which is prohibited usually for stream cipher; it is the context for block ciphers). Then, a stream cipher in a narrower sense (i.e. exor with PN) is easily broken by known-plain text attack, since the PN sequence is recovered by taking exor. In a block cipher, it is required that E_i (and D_i) are difficult to guess from an (even huge) number of pairs $(B, E_i(B))$ where B can be chosen (i.e. chosen plain text attack). Fubuki is designed to have this type of resistance.

5. Description of Fubuki

5.1. **Overview.** Fubuki prepares four wordwize PEFs

`empr`, `emer`, `emps`, `emes`

and four inter-word PEFs

`ma`, `mem`, `ome`, `eme`

and one PEF `vert_rotate`.

One round of Fubuki consists of three stages: choose one of the four wordwise PEFs and apply it to the plain block, then choose one of the four inter-word PEFs, then apply `vert_rotate`. In C-like notation, it is described as

```
c = pseudorandom_two_bits();
switch (c) {
case 0: crypt_empr(msgbuf); break;
case 1: crypt_emer(msgbuf); break;
case 2: crypt_emps(msgbuf); break;
case 3: crypt_emes(msgbuf); break;
}

c = pseudorandom_two_bits();
switch (c) {
case 0: crypt_ma(msgbuf); break;
case 1: crypt_mem(msgbuf); break;
case 2: crypt_ome(msgbuf); break;
case 3: crypt_eme(msgbuf); break;
}
crypt_vert_rotate(msgbuf);
```

Here, `msgbuf` is an array of block size, and each PEF rewrites this array. The parameters are generated in each of PEF by calling MT, so not visible in this description. This round is iterated for `Iteration` times (which is 4 in default case, for 128 bit blocks).

The two-bits pseudorandom integers are generated as follows. We use C-language-like notation.

```
genrand_tuple_int32(func_choice, 4);
func_choice[2] *= (func_choice[0] | 1);
func_choice[3] *= (func_choice[1] | 1);
func_choice[0] ^= (func_choice[3] >> 5);
func_choice[1] ^= (func_choice[2] >> 5);
```

Here, the first function fills four PNs into the array `func_choice`. The next four transformations mix these four words. The notation `*=` is to multiply the right hand side to the left and write to the left, `^=` is similar operation with exclusive-or, `|` is bitwise OR operation, `>> 5` is bit-shift-to-right by 5 bits.

Fubuki uses the 128 bits in this array for function choice: first use the most significant two bits of `func_choice[0]`, then next two bits of `func_choice[0]`, and so on. Thus, if Iteration=4, then most significant 16 bits of `func_choice[0]` are used to choose PEFs 2×4 times.

5.2. **Primitive encryption families.** To make the explanation and the implementation simpler, we choose one uniform parameter space $\mathcal{P}$ for all PEFs. Let t be the number of words in the block (typically four, and assumed to be a power of 2). Then, $\mathcal{B} := W^t$, and we set

$$\mathcal{P} := W^t \times \{1, 2, \ldots, t-1\}.$$

We denote an element of $P \in \mathcal{P}$ and $B \in \mathcal{B}$ by

$$P = (p_0, p_1, \ldots, p_{t-1}, \texttt{jump}), \quad B = (b_0, b_1, \ldots, b_{t-1}).$$

Each of b_i is considered as a variable of wordsize.

5.3. **Wordwise PEFs.** Each of wordwise PEFs is described as follows. Again, t is the number of the words in a block.

The block B is transformed as follows. Let $j = 0$. The first operation is

$$b_j \leftarrow b_j \oplus p_j.$$

Note that its inverse is itself.

Then, multiply a constant

$$b_j \leftarrow b_j \times c_j \mod 2^{32}.$$

The constant c_j should be (multiplicatively) invertible modulo 2^{32}, i.e., should be odd. Their inverses are necessary in decoding, and would be time-consuming if they were computed during the decoding process. So, before starting encryption, Fubuki prepares 32 pseudo-randomly chosen 32-bit integers $m_0, m_1, \ldots, m_{31}$, using MT, and store them in an array `mult_table`. Before decoding, Fubuki computes their inverses $m'_0, m'_1, \ldots, m'_{31}$, and store them in an array `inv_table`.

These multipliers m_k $(k = 0, \ldots, 31)$ are the first 32 outputs of the initialized MT, but by bit-operations we set the least significant four bits of m_k to 1011 for k even, and to 0111 for k odd. Moreover, the $((k \mod 8) + 1)$-st bit and $((k \mod 8) + 2)$-nd bit of m_k are set to 1 and 0, respectively. This is to avoid too trivial multipliers like 1 or $2^{32} - 1$.

The constant c_j is chosen from these multipliers by putting

$$c_j := m_{k_j}, k_j = (p_{j+\ell \bmod t} >> (32-5)).$$

Here, ℓ is a constant and $>> (32-5)$ means right-shift by $32-5$ bits. For `empr`, `emer`, `emps`, `emes`, the constant ℓ is 1, 2, 2, 3, respectively.

Such a multiplication diffuses information of bits in b_j from right to left. To force the diffusion in the inverse direction, Fubuki prepares 32 pseudorandomly chosen 32-bit constants $a_0, a_1, \ldots, a_{31}$ using MT, stored in an array `add_table` of 32 words. This array is filled with the next 32 outputs of MT after setting m_k's. To avoid trivial constants, apply the following operation:

```
for (i=0; i< 32; i++) {
    u32 s;
    s = (i * 1103515245 + 12345) & 31;
    s ^= (s >> 2);
    add_table[i] <<= 5;
    add_table[i] |= s;
}
```

That is, the content of `add_table[i]` is shifted to left by five bits (the notation `<<= 5`), and the least five bits are set to be s_i which is given by

$$s_i \leftarrow (1103515245i + 12345) \mod 32,$$
$$s_i \leftarrow s_i \oplus (s_i >> 2),$$

for $i = 0, \ldots, 31$. (The notation & denotes bitwise-AND, so & 31 is taking modulo 32.) This is just to make the correspondence $i \to s_i$ a complicated permutation on $\{0, 1, \ldots, 31\}$. Then, apply the operation

$$b_{(j+\texttt{jump}) \bmod t} \leftarrow b_{(j+\texttt{jump}) \bmod t} \,\square\, (\texttt{add\_table}[b_j >> (32-5)]), \tag{4}$$

where $\square$ is plus (modulo 2^{32}) for `empr` and `emps`, and exclusive-or for `emer` and `emes`. Here, `jump` is a component of the parameter. Each PEF is designed to rewrite b_j depending on the information in $b_{j-\texttt{jump} \bmod t}$.

The parameter `jump` is not randomly chosen: `jump` is set to 1 before encoding each block, and after executing each PEF, it is rewritten by

$$\texttt{jump} \leftarrow \texttt{jump} \times 2;$$
$$\text{if } \texttt{jump} \geq t \text{ then } \texttt{jump} \leftarrow 1.$$

Thus, jump moves 1, 2, 4, 8,..., and if it becomes more than or equal to t, it is set to 1. This is to scatter the information of one word to the other words in the block as quick as possible. Partial sum of 1, 2, 4,... represents any integer, so after $\log_2(t)$ times iteration of PEFs, the information of one word tends to be passed to all the other words in the block.

A shortcoming of a feedback (4) is that there are only 32 different patterns may occur. However, the most significant bits of b_j gather the information of all the lower bits by multiplying c_j, so a change of one bit in b_j tends to be reflected in the choice of an element in the array `add_table`, so having good bit-diffusion property.

The final operation of wordwise PEF is "rotation" or "shift". Using p_j, obtain pseudorandom integer between 16 and 23 by the following bit-operation

$$s_j \leftarrow ((p_j >> (32-4))|\texttt{0x10})\&\texttt{0x17}, \tag{5}$$

where `0x` denotes that the following number is hexadecimal. Then, the "rotation" operation

$$b_j = ((\sim b_j) << (32 - s_j))|(b_j >> s_j);$$

is computed. This is rotate to the right by s bits, but the rotated s bits are reversed. The unary operator $\sim$ means the bit reverse.

The "shift" operation is

$$b_j \leftarrow b_j \oplus ((\sim b_j) >> s_j). \tag{6}$$

Since s_j is not less than 16 and a word is 32-bit, the inverse of this operation is itself.

The "rotate" is chosen for `empr` and `emer`, and the "shift" is chosen for `emps` and `emes`.

The above operations (i.e. the operations stated in this subsection) are applied for $j = 0, 1, \ldots, t-1$ in this order. This is the description of four wordwise PEFs.

5.4. **Inter-word PEFs.** Wordwize PEFs are mainly to mix the information of each word, except for the operation involving `jump`, which passes (only) 5-bit of information to another word.

Inter-word PEFs `ma`, `mem`, `ome`, `eme` are designed to pass more information of each word to the other words.

First we describe `ma`. Let $j = 0$. Put $k := (j - \texttt{jump}) \bmod t$. Then, we apply the Feistel-network-type operation

$$b_j \leftarrow b_j + (b_k \times p_j).$$

Compute a pseudorandom integer s_j by (5), and apply the "shift" operation (6). Iterate this for $j = 0, 1, \ldots, t-1$, in this order. This describes the PEF `ma`. The other three PEFs are as follows.

Let $j = 0$. Put $k := (j - \texttt{jump}) \bmod t$. Generate a pseudorandom integer s between 0 and $t - 1$ such that $s \neq j$, by

$$\begin{array}{c} s \leftarrow p_j >> (32 - \log_2(t)); \\ \text{if } s = j \text{ then } s \leftarrow ((s-1) \bmod t). \end{array} \tag{7}$$

Then, in the case of `mem`, apply the operation

$$\begin{array}{l} b_j \leftarrow (b_j \oplus (b_k \times b_s)) - p_j \\ b_j \leftarrow b_j \oplus (b_j >> 16). \end{array}$$

In the case of `ome` or `eme`, apply

$$\begin{array}{l} b_j \leftarrow (b_j \oplus (b_k \times (b_s \Box p_j)) \\ b_j \leftarrow b_j \oplus (b_j >> c), \end{array}$$

where $\Box$ =bitwise-OR and $c = 16$ for `ome` and $\Box = \oplus$ and $c = 17$ for `eme`. Iterate this for $j = 0, 1, \ldots, t-1$, in this order. This completes the description of inter-word PEFs.

5.5. **crypt_vert_rotate.** This PEF is to cut off the incidence relation among bits in each word. Put $k := 2 * (p_0 + p_{t-1}) + 1$ (modulo 2^{32}). Put

$$\texttt{jump\_odd} := (\texttt{jump} - 1)|1,$$

which is the largest odd integer not exceeding `jump`. We consider B to be $t \times 32$ matrix with components 0-1, and permute each row vector of B selected by the bit mask k (i.e., if n-th bit of k is 1, then the n-th row is selected). The permutation of a row vector is rotation with lag `jump_odd`. Namely, a t-dimensional row vector ${}^t(x_0, x_1, \ldots, x_{t-1})$ is mapped to ${}^t(x_{-o}, x_{1-o}, \ldots x_{t-1-o})$, where $o = \texttt{jump\_odd}$ and the subscripts are considered modulo t. The `jump_odd` is forced to be odd, since it is faster to compute a cyclic permutation.

5.6. **Security of Fubuki.** Heuristically, one round of Fubuki has much better bit-diffusion property than one round of AES. Each of steps in one round in AES has some corresponding steps in Fubuki: ByteSub and ShiftRow in AES are replaced by wordwise PEFs (multiplication plus feedback from right-to-left (4)), MixColumn is included in inter-word PEFs and AddRoundKey appears in every PEFs where the parameters p_i are added, exor-ed, or subtracted. The 19937 bits of internal state of MT seems to make time-memory-trade-off attacks infeasible. Although, we have to say that we need to study more on the security, such as difference propagation property. Experimental results are shown in §6.2 and Appendix B.

	algorithm	initial	encrypt	decrypt
PowerPC 1.33GHz	crypt-mt	256601	29	30
	fubuki	414029	204	407
	rijndael-alg-fst	2793	41	38
	rijndael-alg-ref	33117	1068	1063
PentiumM 1.4GHz	crypt-mt	279123	20	19
	fubuki	489662	133	256
	rijndael-alg-fst	3192	40	41
	rijndael-alg-ref	85253	916	916

Table 1. execution time (number of cycles per byte)

$\Delta_{\text{fubuki,1}}$	$\Delta_{\text{rijndael,1}}$
00100010111011011111001100110011	00111111000000000000000000000000
11101111001111111100110100011100	00000000000000000000000001010011
00101101111110010000000100100010	00000000000000001111000100000000
10100001000101110110011111100101	00000000010100000000000000000000

Table 2. Differential of one-round Fubuki and two-round AES

6. Comparison with AES

6.1. **Speed and memory.** Table 1 lists the approximate number of CPU cycles consumed to (1) setup keys, (2) encrypt one byte, (3) decrypt one byte, for four stream ciphers CryptMT, Fubuki, optimized AES (rijndael-alg-fst.c) and reference AES (rijndael-alg-ref.c) [1]. Rough estimate of the size of the working area is 2.5K bytes, 3K bytes, 10.5K bytes, and 1.4K bytes, respectively.

6.2. **Bit-diffusion.** Let $E_0 : \mathcal{B} \to \mathcal{B}$ be one round of Fubuki encryption, with both the key and the initial value are 128 bits of zeroes. To grasp the bit-diffusion property of E_0, we compute

$$\Delta_{\text{fubuki,i}} := E_0(\epsilon_i) \oplus E_0(0),$$

where ϵ_i is a 128-bit vector with all zero components except for the i-th bit being 1. For comparison, we compute

$$\Delta_{\text{rijndael,i}} := A_0(\epsilon_i) \oplus A_0(0),$$

where A_0 is the 2-round AES with randomly chosen key.

The result for $i = 1$ is described in Table 2. The bit patterns suggest that one round of Fubuki seems to have quicker bit-diffusion property than two rounds of rijndael. Similar results are observed for all $1 \leq i \leq 128$.

Appendix A. Period of CryptMT

In this appendix, we shall prove the following.

Theorem A.1. The period of the output sequence of CryptMT is $P := 2^{19937} - 1$. Moreover, every bit of the sequence has period P.

This may seem to be obvious, but we think not so.

Let $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \ldots$ be the output of MT. MT has the following 623-dimensional equidistribution property.

Lemma A.2. Any bit pattern of (32×623) bits occurs the same (actually two) times in the form of the tuple

$$(\mathbf{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_{i+622})$$

when we move $i = 0, 1, \ldots, P-1$. There is an exception: the all-zero pattern occurs only once.

In the following proof, we use only this property and the fact that P is a prime.

Let us define $\mathbf{x}'_i$ to be $\mathbf{x}_i$ with the least significant bit set to 1. We denote by $\mathbb{Z}/2^{32}$ the residue ring modulo 2^{32} identified with 32-bit integers, and by $(\mathbb{Z}/2^{32})^\times$ its multiplicative group, identified with 32-bit odd integers. In the following, multiplication of 32-bit integers are taken in this group, i.e. modulo 2^{32}.

Let us define 32-bit word sequence a_i by

$$a_{i+1} = a_i \mathbf{x}'_i, \quad (a_0 = \text{some odd integer}).$$

The output sequence of CryptMT is the sequence of the most significant 8 bits of a_i.

Corollary A.3. Any element of $((\mathbb{Z}/2^{32})^\times)^{623}$ occurs the same times in the form of the tuple $(\mathbf{x}'_i, \mathbf{x}'_{i+1}, \ldots, \mathbf{x}'_{i+622})$ when we move $i = 0, 1, \ldots, P-1$, except for $(1, 1, \ldots, 1)$ which occurs once less often.

Any element of $(\mathbb{Z}/2^{32})^\times$ occurs same times in the form of $\mathbf{x}_i$ for $i = 0, 1, \ldots, P-1$.

Corollary A.4.

$$\prod_{i=0}^{P-1} \mathbf{x}'_i = 1.$$

Proof. By cancellation of x and x^{-1}, for any abelian group G we have

$$\prod_{x \in G} x = \prod_{x^2=1} x.$$

In the case of $G = (\mathbb{Z}/2^{32})^\times$, the elements of order two are exactly $x = 1, -1, 2^{32-1} - 1, 2^{32-1} + 1$, and their product is 1. The conclusion is deduced from the above corollary. □

Lemma A.5. The period of a_i divides P.

Proof. This is because

$$a_{i+P} = (\prod_{i=0}^{P-1} \mathbf{x}_i) a_i,$$

which is a_i by the above corollary. □

Since P is a (Mersenne) prime, in particular, every bit of a_i has period 1 or P. Thus, to show the theorem, we only need to show that its period is not 1.

Lemma A.6. Define a function

$$\varphi : ((\mathbb{Z}/2^{32})^\times)^{624} \to ((\mathbb{Z}/2^{32})^\times)^{623}$$

by

$$(b_0, \ldots, b_{623}) \mapsto (b_1/b_0, b_2/b_1, \ldots, b_{623}/b_{622}).$$

Define sets A and X by

$$A := \{(a_i, a_{i+1}, \ldots, a_{i+623}) | i = 0, 1, 2, \ldots\},$$

$$X := \{(x_i, x_{i+1}, \ldots, x_{i+622}) | i = 0, 1, 2, \ldots\}.$$

Then, the restriction $\varphi|_A : A \to X$ is surjective.

Proof. Because $(a_i, a_{i+1}, \ldots, a_{i+623})$ is mapped to $(x_i, x_{i+1}, \ldots, x_{i+622})$. □

Now we prove the theorem. Suppose that ℓ-th bit of a_i has period 1. Then, each a_i is contained in the subset of $(\mathbb{Z}/2^{32})^\times$ whose ℓ-th bit coincides with that of a_i. There are 2^{30} such odd integers, so we have

$$\#(A) \le 2^{30 \times 624}.$$

On the other hand, by Corollary A.3, we have

$$\#(X) = 2^{31 \times 623},$$

which is larger than $\#(A)$, contradicting to the existence of a surjection $A \to X$. This shows that ℓ-th bit has period > 1, but it divides the prime P, hence $= P$.

Remark A.7. The lemma A.6 claims that $\#(A) \ge 2^{31 \times 623}$, where $A \subset ((\mathbb{Z}/2^{31})^\times)^{624}$. This suggests that a_i would be fairy well equi-distributed in a high-dimensional cube, and so would the output of CryptMT.

Remark A.8. The above remark is valid for

(1) any mother generator with high-dimensional equidistribution property, and
(2) any binary operation $\circ$

$$a' := a \circ x$$

which is invertible, i.e. $x \mapsto a \circ x$ is bijective. In our case, $\circ$ is the multiplication.

Appendix B. Experimental comparison of Fubuki and AES

We tested the bit-diffusion property of Fubuki and AES with small number of rounds, as follows. This is a typical differential attack to block ciphers.

Let $E : \mathcal{B} \to \mathcal{B}$ be an encryption function. Let $\mathcal{B}' \subset \mathcal{B}$. We uniformly randomly generate $B \in \mathcal{B}'$, and compute the difference $\Delta_E(B) := E(B \oplus \epsilon_1) \oplus E(B)$ (ϵ_1 is a block whose bits are all zero except for the first bit). The resulting 128 bits = 4 words are considered to be 4×4 matrix of 8-bit integers. Then the (i, j)-component $\Delta^E_{ij}(B)$ of $\Delta_E(B)$ is a random variable with values in $0, 1, \ldots, 255$, which should ideally be uniform.

We generate B randomly N times. Then we count the number of occurrence of an integer k ($0 \le k \le 255$) as $\Delta^E_{ij}(B)$, denoted by $F^E_{ij}(N)[k]$ (F for frequency). If $F^E_{ij}(N)[k] = 0$ then k did not appear in the (i, j) component of $\Delta^E(B)$ for N times. Ideally the expectation of $F^E_{ij}(N)[k]$ should be $N/256$. We take the following normalization:

$$f^E_{ij}(N)[k] := F^E_{ij}(N)[k] \times 256/N.$$

We compute these statistics, and lists the minimum and the maximum of $f^E_{ij}(N)[k]$ for $k = 0, \ldots, 255$.

3-round AES			
$0.73700, 1.20090$	$0.68740, 1.21830$	$0.78480, 1.26710$	$0.73810, 1.24930$
$0.66180, 1.23490$	$0.78750, 1.28410$	$0.75200, 1.24890$	$0.73110, 1.20510$
$0.77420, 1.28260$	$0.74730, 1.23530$	$0.73150, 1.21340$	$0.67320, 1.21810$
$0.75130, 1.23510$	$0.73240, 1.19540$	$0.68640, 1.21500$	$0.77710, 1.29130$
4-round AES			
$0.97610, 1.02900$	$0.97460, 1.02790$	$0.97670, 1.02280$	$0.96700, 1.02820$
$0.96750, 1.03410$	$0.96760, 1.02740$	$0.97730, 1.03680$	$0.97410, 1.02370$
$0.97660, 1.02330$	$0.97060, 1.03260$	$0.97070, 1.02670$	$0.96830, 1.03220$
$0.97290, 1.03590$	$0.96650, 1.03000$	$0.97220, 1.02920$	$0.97600, 1.02280$
1-round Fubuki			
$0.51770, 1.66800$	$0.69160, 1.54620$	$0.24140, 3.50450$	$0.38440, 1.59600$
$0.01450, 4.30050$	$0.88450, 1.10840$	$0.43290, 1.44460$	$0.43300, 2.35560$
$0.04200, 1.96300$	$0.00000, 3.98790$	$0.00000, 3.14730$	$0.12590, 2.16670$
$0.97650, 1.02430$	$0.97670, 1.02490$	$0.97740, 1.02910$	$0.97120, 1.03330$
2-round Fubuki			
$0.97480, 1.02720$	$0.97790, 1.02440$	$0.97250, 1.02920$	$0.97340, 1.02710$
$0.97000, 1.03010$	$0.96790, 1.02700$	$0.97450, 1.02540$	$0.97270, 1.02880$
$0.97600, 1.02530$	$0.97650, 1.03050$	$0.97360, 1.02970$	$0.96860, 1.03170$
$0.97480, 1.02900$	$0.96770, 1.02820$	$0.96690, 1.02770$	$0.97260, 1.03390$

Table 3. Differentials of 3-round AES, 4-round AES, 1-round Fubuki and 2-round Fubuki. Lists of the normalized frequency (minimum, maximum) for $N = 2,560,000$ times sampling

.

Table 3 shows the result of the tests. We selected 3-round AES, 4-round AES, 1-round Fubuki and 2-round Fubuki as ciphers. We choose $\mathcal{B}'$ to be the blocks whose first word is arbitrary and the rest three words are zero. We take $N = 2560000$ such sample blocks, and compute the difference caused by adding ϵ_1 as above. The table lists the minimum and and the maximum normalized frequencies. For example, the first pair $(0.73700, 1.20090)$ in the table of 3-round AES shows that there is some k, $k = 0, 1, 2, \ldots, 255$, for which $(1, 1)$-coordinate byte of the block-difference takes value k for $0.73700 \times 2560000/256$ times. This is the least frequency among 256 possible values as for the (1,1)-coordinate.

The table shows that 3-round AES is still weak, since for 4-round AES, the frequencies are much different. Although not listed, 2-round AES has many (0, 256). For 1-round Fubuki, there are several zeroes as minimum-frequencies. This implies that there are some impossible 8-bit patterns for that coordinate. Such bias seems to be eliminated in 2-round Fubuki.

Although the result is omitted, experimental results for larger rounds are similar to that for 4-round AES (and 2-round Fubuki).

References

[1] AES lounge: http://www.iaik.tu-graz.ac.at/research/krypto/AES/
[2] Bernstein, D. J. Cache-timing attack on AES
http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[3] Knuth, D. E. The Art of Computer Programming. Vol. 2. Seminumerical Algorithms 3rd Ed. Addison-Wesley, Reading, Mass., (1997).
[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. and Nishimura, T. Mersenne Twister Homepage. http://www.math.sci.hiroshima-u.ac.jp/˜m-mat/emt.html
[6] Tusnoo, Y., Saito, T., Suzaki, T., Shigeri, M., and Miyauchi, H. Cryptanalysis of DES implemented on computers with cache, in Cryptographic hardware and embedded systems–CHES 2003, Springer-Verlag, Berlin (2003), 62–76.

Department of Mathematics, Hiroshima University, Hiroshima 739-8526, JAPAN
E-mail address: `m-mat@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`

Department of Mathematics, Hiroshima University, Hiroshima 739-8526, JAPAN
E-mail address: `saito@math.sci.hiroshima-u.ac.jp`