MUGI Pseudorandom Number Generator

Specification Ver. 1.3

Hitachi, Ltd.

2002. 5. 14

Contents

1 Introduction

This documentation gives a description of **MUGI** pseudorandom number generator. MUGI has two independent parameters. One is 128-bit secret key, and another is 128-bit initial vector. The initial vector can be public.

The document is organized as follows: In Section 2 we show the design rationale of MUGI. Next we give some notations and some fundamental knowledges in Section 3. In Section 4 we describe the specification of MUGI in detail. At last we give some usage notes in Section 5.

2 Design Rationale

MUGI is a pseudorandom number generator (PRNG) designed for using as a stream cipher. The design is aimed to be suitable for both of software and hardware.

Nowadays the design of a block cipher is well sophisticated so that it can be suitable for any platforms and achieve good performances. On the other hand almost all of stream ciphers are dedicated to a special implementation. In addition some algorithm suitable for software is not well evaluated. We are obliged to conclude that the design of stream ciphers suitable for software is not so sophisticated as one of block ciphers at present.

In this situation we pay attention to PANAMA [DC98]. PANAMA was designed by J. Daemen and C. Clapp in 1998, and is a cryptographic module that can be used both as a hash function and a stream cipher.

The designers of PANAMA did not fasten upon the design using linear feedback shift registers (LFSR), which were main stream in the design of stream ciphers, but the principle design of block ciphers. This implies the evaluation techniques are applicable to PANAMA. Furthermore its design strategy is simple and has generality. So we can design a PRNG similar to PANAMA easily. On the other hand the design of PANAMA is unprecedented so that the security of PANAMA is not evaluated enough at present.

The design of our PRNG MUGI is similar to PANAMA. Additionally we aim to evaluate its security as well as possible. See [Eval] in reference to the security evaluation of MUGI.

As a result, MUGI achieves high performance as well as AES [FIPS-197].

Especially the hardware implementation is excellent. On the other hand we believe that the security is evaluated enough in [Eval].

In the following of this section, first, we roughly describe the structure of PANAMA and MUGI in 2.1. Then we mention the component of MUGI in 2.2

2.1 PANAMA-like keystream generator

Generally the principal part of a PRNG is a set $(\mathcal{S}, \mathcal{F}, f)$ which consists of an internal state $\mathcal{S}$, its update function $\mathcal{F}$, and the output filter f which abstracts the output sequence from the internal state $\mathcal{S}$. Especially we call the set $(\mathcal{S}, \mathcal{F})$ as an **internal-state machine**. In addition we call a step that the update function is applied as a **round**. $\mathcal{S}^{(t)}$ refers to the internal state at round t.

In the case of PANAMA, the internal state is divided into two parts, the state a and the buffer b. The update function of PANAMA is divided in proportion to the internal state (see Figure 1). Note that each update function uses another part of the internal state as a parameter. We denote the update function of the state a and the buffer b as ρ and λ function respectively.

It is distinct in the update function of PANAMA that the function ρ has a SPN structure. It is similar to a block cipher's round function. On the other hand the function λ is a simple linear transformation. The output filter f abstracts about half bits of the state a for each round.

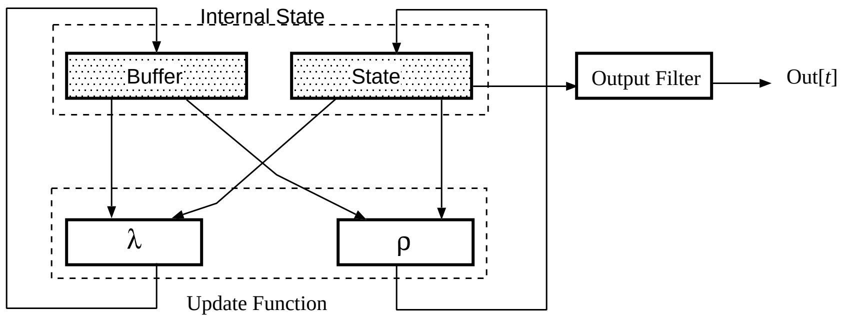

Figure 1: The scheme of PANAMA-like keystream generator

We call a PRNG which satisfies above characteristics as PANAMA-like keystream generator (PKSG). We shape into the definition of PKSG as below:

Definition 1 *Consider that an internal-state machine consists of the internal state (the state a and the buffer b) and their update functions (ρ and λ). The keystream generator which consists of $(a, b), (\rho, \lambda)$, and output filter f is called* PANAMA***-like keystream generator*** *if it satisfies following conditions:*

(1) ρ includes an SPN transformation and uses parts of the buffer b as a parameter.

$$a^{(t+1)} = \rho(a^{(t)}, b^{(t)}).$$

(2) λ is a linear transformation and uses a part of the state a as a parameter.

$$b^{(t+1)} = \lambda(b^{(t)}, a^{(t)}).$$

(3) f outputs a part of the state a (usually no more than $1/2$).

2.2 Selection of components

We make a point of reusing existing good articles on design of MUGI. As a result we use some component of AES [FIPS-197], which is well evaluated. For example the substitution table S-box and the linear transformation is same as AES. Though the design of PKSG is still alternative, this selection should make MUGI more secure.

3 Preliminaries

In this section we give some notations and preliminary knowledge.

3.1 Notations

$\oplus$	bitwise XOR
$\wedge$	bitwise AND
$\|\|$	concatenation of two strings
$>>> n$	rotation of n bits to right (in 64-bit register)
$<<< n$	rotation of n bits to left (in 64-bit register)
`0x`	prefix meaning hexadecimal integer

3.2 Data Structure

The elemental-data size of MUGI is 64 bit, called a **unit**. Embedding byte data into a 64-bit word, we adopt big-endian. For example 8-byte input data $x_0, \ldots, x_7$ is stored into one unit as follows:

$$a = [\text{MSB}] \quad x_0||x_1||\ldots||x_7 \quad [\text{LSB}],$$

where [MSB] and [LSB] represent the positions of the most significant byte and the least significant byte, respectively.

On the other hand the output key stream is given as a unit data.

The j-th byte (from the most significant side) of unit a is denoted by a_j. When we use plural subscript, the first subscript specifies unit position and the second subscript specifies the byte position. For instance in the sequence consists of unit data $B = (b_i)_i$, $b_{i,j}$ means the j-th byte in the i-th unit.

The higher and lower 32-bit data of a unit is denoted by the subscripts "H" and "L", e.g. $(a_H||a_L) = a$.

3.3 Finite Field GF(2^8)

3.3.1 Data Expression

MUGI uses some operations in finite field GF(2^8). A finite field has many different representations. We fixed a characteristic polynomial and represent the element of GF(2^8) by a polynomial.

First of all we define the finite field GF(2^8) as GF(2^8) = GF(2)[x]/($\varphi(x)$), there the polynomial $\varphi(x)$ is given as follows:

$$\varphi(x) = x^8 + x^4 + x^3 + x + 1 \leftrightarrow \texttt{0x11b}.$$

Any element in GF(2^8) is represented by 1-variable polynomial whose coefficients are in GF(2) (i.e., the coefficients are in $\{0, 1\}$) and the degree is no more than 7. The binary representation (for implementation) is given by 8-bit data. The 8-bit string $b_7||b_6||b_5||b_4||b_3||b_2||b_1||b_0$ is associated to

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0.$$

For example a byte data `0x57` is associated to the bit string `0101 0111`, $x^6 + x^4 + x^2 + x + 1$.

3.3.2 Addition

The sum of two polynomials over GF(2^8) is the polynomial whose coefficients are given by the sum of corresponding coefficients modulo 2. In other words the addition is caluculated by bitwise XOR. For example, the sum of `0x57` and `0xa3` is calculated as follows:

$$\begin{aligned} \texttt{0x57} + \texttt{0xa3} &= (x^6 + x^4 + x^2 + x + 1) + (x^7 + x^5 + x + 1) \\ &= x^7 + x^6 + x^5 + x^4 + x^2 \\ &\leftrightarrow \texttt{0xf4}. \end{aligned}$$

3.3.3 Multiplication

The multiplication in GF(2^8) can be divided into two steps.

Firstly we give the multiplication of any element $f(x) = \sum a_i x^i$ and x as follows:

$$x \cdot f(x) = \sum b_i x^{i+1} \bmod \varphi(x).$$

For example, the multiplication of `0x02` and `0x87` is calculated as follows:

$$\texttt{0x02} \cdot \texttt{0x87} = x \cdot (x^7 + x^2 + x + 1)$$

$$\begin{aligned} &= x^8 + x^3 + x^2 + x \\ &= (x^4 + x^3 + x + 1) + x^3 + x^2 + x \\ &= x^4 + x^2 + 1 \\ &\leftrightarrow \texttt{0x15} \end{aligned}$$

$x^i \cdot f(x)$ for any i can be calculated by iterative application of above definition.

The multiplication $f \cdot g$ of any two element $f(x) = \sum a_i x^i$, $g(x) = \sum b_i x^i$ in GF(2^8) is defined as follows:

$$f \cdot g(x) = \sum_{i=0}^{14} \sum_{j=0}^{i} (a_j \wedge b_{i-j}) x^i \bmod \varphi(x)$$

3.3.4 Inverse

For $f, g \in \mathrm{GF}(2^8)$, we call g an inverse of f and denote $g = f^{-1}$ if there are $a, b \in \mathrm{GF}(2^8)$ satisfing the following equation:

$$f \cdot a + g \cdot b = 1 \bmod \varphi(x).$$

It is well known that any element in any finite field except 0 has its inverse. In the case of GF(2^8) the inverse of a is given by $a^{-1} = a^{254}$.

4 Specification

In this section we give a description of MUGI. As we mention in Section 2 any PRNG is described as the combination of an internal-state machine and an output filter.

We describe the internal state of MUGI in 4.3 and the update function in 4.4 at first. We show the detail description of the components of the transition in 4.7. Then we mentioned the initialization in 4.5 and the random number generation in 4.6.

4.1 Outline

MUGI is a PRNG with an 128-bit secret key K (secret parameter) and an 128-bit initial vector I (public parameter). It generates 64-bit length random bit string for each round. The outline of the algorithm is as follows:

Input: Secret key K, Initial vector I, Output size n (units)

Output: Random number sequence $Out[i]$ $(1 \leq i \leq n)$

Algorithm
Initialization

Step 1. First set the secret key K into the state a. Then initialize the buffer b by means of ρ.

Step 2. Add the initial vector I into the state a and the initialize state a by means of ρ.

Step 3. Mix whole internal state by means of the update function.

Random number generation

Step 4. Run n rounds update function and output a part of the internal state (64 bit) for each round.

Now we explain each of above in detail.

4.2 Input

MUGI has two inputs. One is an 128-bit secret key K and the other is an 128-bit initial vector I. I is a public parameter. The higher and lower units of K are denoted by K_0 and K_1 respectively. In the same manner I_0 and I_1 are used in this document.

4.3 Internal State

4.3.1 State

The state a consists of 3 units. Each of them is denoted by a_0, a_1, a_2 in rotation, i.e.

$$a = \text{[Higher]} \quad a_0||a_1||a_2 \quad \text{[Lower]}.$$

4.3.2 Buffer

The buffer n consists of 16 units. Each of them is denoted by $b_0, \ldots, b_{15}$ in rotation in the same manner as the state a.

4.4 Update Function

In general the update function of PKSG is described as a combination of ρ and λ, (the update functions of the state a and the buffer b), each of which uses another internal state as a parameter. In other words the update function *Update* of whole internal state is described as follows:

$$(a^{(t+1)}, b^{(t+1)}) \quad = \quad Update(a^{(t)}, b^{(t)}) \quad = \quad (\rho(a^{(t)}, b^{(t)}), \lambda(b^{(t)}, a^{(t)})).$$

In the followings we explain ρ and λ of MUGI.

4.4.1 Rho

ρ is the update function of the state a. It is a kind of target heavy Feistel structure with two F-functions (Figure 2) and uses the buffer b as a parameter. The function ρ is described as follows:

$$\begin{aligned}
a_0^{(t+1)} &= a_1^{(t)}, \\
a_1^{(t+1)} &= a_2^{(t)} \oplus \mathrm{F}(a_1^{(t)}, b_4^{(t)}) \oplus C_1, \\
a_2^{(t+1)} &= a_0^{(t)} \oplus \mathrm{F}(a_1^{(t)}, b_{10}^{(t)} <<< 17) \oplus C_2,
\end{aligned}$$

C_1, C_2 in the equations above are constants. The F-function of MUGI reuses

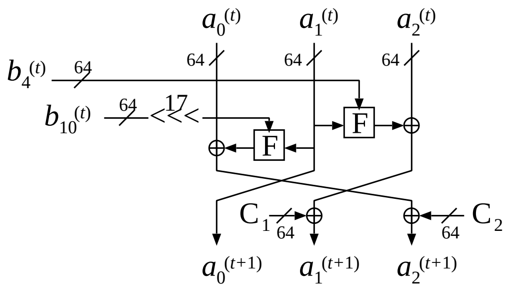

Figure 2: ρ-function

the components of AES (S-box, MDS). We show the detail description in 4.7.3.

4.4.2 Lambda

The function λ is the update function of the buffer b and used a part of the state a as a parameter. λ is linear transformation of b and is described as follows:

$$\begin{aligned} b_j^{(t+1)} &= b_{j-1}^{(t)} \quad (j \neq 0, 4, 10), \\ b_0^{(t+1)} &= b_{15}^{(t)} \oplus a_0^{(t)}, \\ b_4^{(t+1)} &= b_3^{(t)} \oplus b_7^{(t)}, \\ b_{10}^{(t+1)} &= b_9^{(t)} \oplus (b_{13}^{(t)} <<< 32). \end{aligned}$$

4.5 Initialization

The initialization of MUGI is divided into three steps. Firstly initialize the buffer b with a secret key K, secondly initialize the state a with an initial vector I, and mix whole internal state at last.

In the first step we set the secret key K into the state a as follows:

$$\begin{aligned} a_0 &= K_0, \\ a_1 &= K_1, \\ a_2 &= (K_0 <<< 7) \oplus (K_1 >>> 7) \oplus C_0. \end{aligned}$$

C_0 in above equation is a constant (see 4.7.4). Then iterate running only ρ and put a part of each $a^{(t)}$ into the buffer b as follows:

$$b_{15-i} = (\rho^{i+1}(a, 0))_0$$

In above equations ρ^i means the i-th iteration of ρ and $\rho(a, 0)$ means the input from b is 0. In other words the data stored into the buffer b is not used for this step.

In the second step the mixed state $a(K) = \rho^{16}(a_0, 0)$ and the initial vector I are required. I is added to the state a as follows:

$$\begin{aligned} a(K, I)_0 &= a(K)_0 \oplus I_0, \\ a(K, I)_1 &= a(K)_1 \oplus I_1, \\ a(K, I)_2 &= a(K)_2 \oplus (I_0 <<< 7) \oplus (I_1 >>> 7) \oplus C_0. \end{aligned}$$

Then the state a is mixed again by 16 rounds iteration of ρ. So the mixed state a is represented as $\rho^{16}(a(K, I), 0)$.

The last step is 16 rounds iteration of whole update function *Update*, i.e.

$$a^{(1)} = Update^{16}(\rho^{16}(a(K, I), 0), b(K)),$$

where the notation $b(K)$ in above equation means the buffer b initialized with the secret key K.

4.6 Random Number Generation

After the initialization MUGI generates 64-bit random number and transforms the internal state at each round. Denote the output at round t as $Out[t]$, then the output is given as below:

$$Out[t] = a_2^{(t)}$$

In other words MUGI outputs the lower 64 bits of the state a at the beginning of the round process.

The processes from the initialization to the random number generation follow Table 1.

Table 1: Time table of MUGI

	Round t	Process	Input	Output
Initialization	-49	Inputting Key	K	–
	$-48, \ldots, -33$	Mixing (by ρ)	–	–
	-32	Inputting IV	I	–
	$-31, \ldots, -16$	Mixing (by ρ)	–	–
	$-15, \ldots, 0$	Mixing (by *Update*)	–	–
Generating bit strings	$1, \ldots$	Outputting and Mixing	–	$Out[t]$

4.7 Components

In this subsection we describe some terms which are used in 4.4 and 4.5 without notice. Especially F-function in 4.4.1 is the main transformation in our PRNG. The F-function adopts 1-round SPN structure and consists of bytewise substitution (denoted S-box) and 4×4 matrix based on GF(2^8). We explain an S-box in 4.7.1, the matrix in 4.7.2, whole construction of the F-function in 4.7.3, and the constants used in MUGI in 4.7.4.

4.7.1 S-box

The bytewise substitution S-box in MUGI is same as one in AES. In other words, the substitution given by S-box is the composition of the inverse $x \rightarrow x^{-1}$ on GF(2^8) and an affine transformation. In the matrix form, the affine transformation of the S-box can be expressed as;

$$\begin{aligned} b' &= S(x) \Leftrightarrow \\ & b = x^{-1}, \\ & \begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \end{aligned}$$

See Appendix A in reference to the substitution table of S-box.

4.7.2 Matrix

The linear transformation of the F-function is the combination of a 4×4 matrix and bytewise shuffling. MUGI uses MDS matrix which is the component of AES. Let M be the matrix and $X = x_0||x_1||x_2||x_3$ be 4 bytes input to M. Then the transformation defined by M is described as follows:

$$M(x) = M \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

The bytewise operation is defined in 3.3.

Note that only `0x01`, `0x02`, and `0x03` are the elements of the matrix. The fact that the multiplication `0x01` $\cdot x$ defines identical map and `0x03` $=$ `0x01` $\oplus$ `0x02` implies that the multiplication `0x02` $\cdot x$ is essentially required in the implementations. Furthermore these multiplications can be implemented by a table lookup. It allows faster implementation than the actual multiplication. See Appendix B in reference to the table for multiplication `0x02` $\cdot x$.

4.7.3 F-function

The F-function is composition of a key addition (the data addition from the buffer), a non-linear transformation using the S-box, a linear transformation using MDS matrix M and byte shuffling (Figure 3). Let denote the input to the F-function as X, the output as Y. Then the F-function is described as follows:

$$Y = F(X, B) \Leftrightarrow$$

$$\begin{aligned}
&O = X \oplus B,\\
&O_0||O_1||O_2||O_3||O_4||O_5||O_6||O_7 = O,\\
&P_i = S(O_i) \quad (0 \le i < 8),\\
&P_H = P_0||P_1||P_2||P_3, \quad P_L = P_4||P_5||P_6||P_7,\\
&Q_H = M(P_H), \quad Q_L = M(P_L),\\
&Q_0||Q_1||Q_2||Q_3 = Q_H, \quad Q_4||Q_5||Q_6||Q_7 = Q_L,\\
&Y = Q_4||Q_5||Q_2||Q_3||Q_0||Q_1||Q_6||Q_7.
\end{aligned}$$

The S-box and the matrix M can be combined in one table lookup on a 32-bit processor [FIPS-197]. It allows the fast implementation.

4.7.4 Constants

There are three constants used in the algorithm of MUGI, C_0 in the initialization, and C_1, C_2 in ρ. These are given as follows:

$$\begin{aligned}
C_0 &= \texttt{0x6A09E667F3BCC908},\\
C_1 &= \texttt{0xBB67AE8584CAA73B},\\
C_2 &= \texttt{0x3C6EF372FE94F82B}.
\end{aligned}$$

These are hexadecimal values of $\sqrt{2}, \sqrt{3}, and \sqrt{5}$ multiplied by 2^{64}.

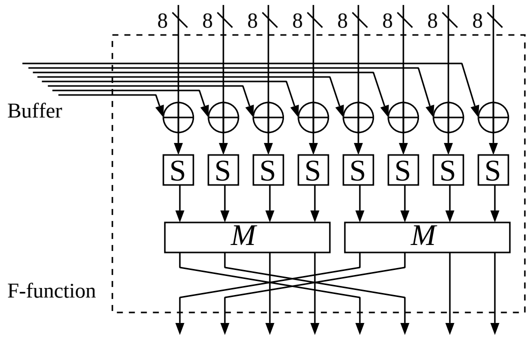

Figure 3: F-function

5 An Encryption Scheme and Remarks

5.1 How to Use Keys and Initial Vectors

In general the output sequence generated by any PRNG is decided by the combination of the secret key K and the initial vector I. So never use an identical combination twice. Especially you must use different initial vector when the secret key is fixed.

5.2 Encryption and Decryption

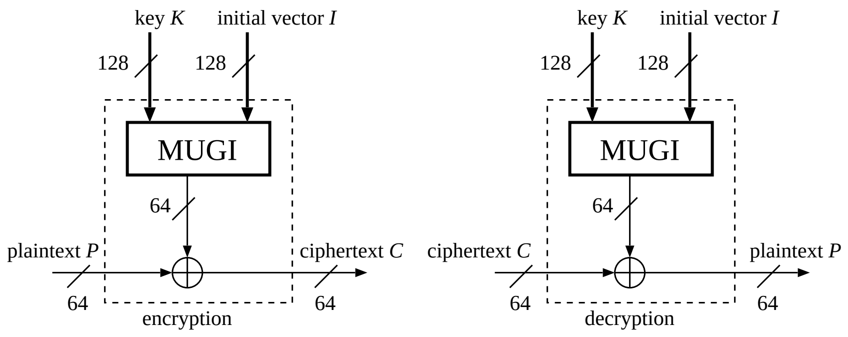

Figure 4: Encryption and decryption using MUGI

MUGI can be used as a stream cipher easily. First, divide the plaintext data into 64-bit blocks. Then XOR them to the output units generated by the secret key K and the initial vector I at each round (see Figure 4). The decryption can be implemented in the same manner.

References

[DC98] J. Daemen, C. Clapp, "Fast Hashing and Stream Encryption with PANAMA," *Fast Software Encryption, 5th International Workshop, FSE'98, Proceedings*, LNCS Vol. 1372, Springer-Verlag, 1998.

[FIPS-197] National Institute of Standards and Technology, Federal Information Processing Standards Publication 197, Advanced Encryption Standard (AES).

[WFT01] D. Watanabe, S. Furuya, K. Takaragi, "The design of key stream generator using F-function of a block cipher," SCIS 2001-6A-4, 2001 (*in Japanese*).

[WFST01a] D. Watanabe, S. Furuya, Y. Seto, K. Takaragi, "A Keystream Generator Suitable for Software Implementation," ISEC 2001-8, 2001 (*in Japanese*).

[WFST01b] D. Watanabe, S. Furuya, Y. Seto, K. Takaragi, "The correlation of the output sequence generated by the Panama-like keystream generator," ISEC 2001-57, 2001 (*in Japanese*).

[Eval] D. Watanabe, S. Furuya, H. Yoshida, K. Takaragi, *MUGI Pseudorandom number generator, Evaluation Report*, 2001, available at `http://www.sdl.hitachi.co.jp/crypto/mugi/index-e.html`.

A S-box

The S-box used in F-function is a substitution table as below:

$$S(x) = \text{Sbox}[x]$$

Sbox[256] = {

```
 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,
0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c,
0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e,
0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68,
0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16  };
```

B The multiplication table for $\texttt{0x02} \cdot x$

The multiplication $\texttt{0x02} \cdot x$ defined in 3.3 is realized by following table:

$$\texttt{0x02} \cdot x = \mathrm{mul2}[x]$$

```
mul2[256] = {
 0x00, 0x02, 0x04, 0x06, 0x08, 0x0a, 0x0c, 0x0e,
0x10, 0x12, 0x14, 0x16, 0x18, 0x1a, 0x1c, 0x1e,
0x20, 0x22, 0x24, 0x26, 0x28, 0x2a, 0x2c, 0x2e,
0x30, 0x32, 0x34, 0x36, 0x38, 0x3a, 0x3c, 0x3e,
0x40, 0x42, 0x44, 0x46, 0x48, 0x4a, 0x4c, 0x4e,
0x50, 0x52, 0x54, 0x56, 0x58, 0x5a, 0x5c, 0x5e,
0x60, 0x62, 0x64, 0x66, 0x68, 0x6a, 0x6c, 0x6e,
0x70, 0x72, 0x74, 0x76, 0x78, 0x7a, 0x7c, 0x7e,
0x80, 0x82, 0x84, 0x86, 0x88, 0x8a, 0x8c, 0x8e,
0x90, 0x92, 0x94, 0x96, 0x98, 0x9a, 0x9c, 0x9e,
0xa0, 0xa2, 0xa4, 0xa6, 0xa8, 0xaa, 0xac, 0xae,
0xb0, 0xb2, 0xb4, 0xb6, 0xb8, 0xba, 0xbc, 0xbe,
0xc0, 0xc2, 0xc4, 0xc6, 0xc8, 0xca, 0xcc, 0xce,
0xd0, 0xd2, 0xd4, 0xd6, 0xd8, 0xda, 0xdc, 0xde,
0xe0, 0xe2, 0xe4, 0xe6, 0xe8, 0xea, 0xec, 0xee,
0xf0, 0xf2, 0xf4, 0xf6, 0xf8, 0xfa, 0xfc, 0xfe,
0x1b, 0x19, 0x1f, 0x1d, 0x13, 0x11, 0x17, 0x15,
0x0b, 0x09, 0x0f, 0x0d, 0x03, 0x01, 0x07, 0x05,
0x3b, 0x39, 0x3f, 0x3d, 0x33, 0x31, 0x37, 0x35,
0x2b, 0x29, 0x2f, 0x2d, 0x23, 0x21, 0x27, 0x25,
0x5b, 0x59, 0x5f, 0x5d, 0x53, 0x51, 0x57, 0x55,
0x4b, 0x49, 0x4f, 0x4d, 0x43, 0x41, 0x47, 0x45,
0x7b, 0x79, 0x7f, 0x7d, 0x73, 0x71, 0x77, 0x75,
0x6b, 0x69, 0x6f, 0x6d, 0x63, 0x61, 0x67, 0x65,
0x9b, 0x99, 0x9f, 0x9d, 0x93, 0x91, 0x97, 0x95,
0x8b, 0x89, 0x8f, 0x8d, 0x83, 0x81, 0x87, 0x85,
0xbb, 0xb9, 0xbf, 0xbd, 0xb3, 0xb1, 0xb7, 0xb5,
0xab, 0xa9, 0xaf, 0xad, 0xa3, 0xa1, 0xa7, 0xa5,
0xdb, 0xd9, 0xdf, 0xdd, 0xd3, 0xd1, 0xd7, 0xd5,
0xcb, 0xc9, 0xcf, 0xcd, 0xc3, 0xc1, 0xc7, 0xc5,
0xfb, 0xf9, 0xff, 0xfd, 0xf3, 0xf1, 0xf7, 0xf5,
0xeb, 0xe9, 0xef, 0xed, 0xe3, 0xe1, 0xe7, 0xe5  };
```

C Test Vector

Example 1.
key[16] = {0}
iv[16] = {0}
output =

```
0xc76e14e70836e6b6, 0xcb0e9c5a0bf03e1e,
0x0acf9af49ebe6d67, 0xd5726e374b1397ac,
0xdac3838528c1e592, 0x8a132730ef2bb752,
0xbd6229599f6d9ac2, 0x7c04760502f1e182,
...
```

Example 2.
key[16] =
{0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f}
iv[16] =
{0xf0, 0xe0, 0xd0, 0xc0, 0xb0, 0xa0, 0x90, 0x80,
0x70, 0x60, 0x50, 0x40, 0x30, 0x20, 0x10, 0x00}
output =

```
0xbc62430614b79b71, 0x71a66681c35542de,
0x7aba5b4fb80e82d7, 0x0b96982890b6e143,
0x4930b5d033157f46, 0xb96ed8499a282645,
0xdbeb1ef16d329b15, 0x34a9192c4ddcf34e,
...
```