Internet Research Task Force R. Tse
Internet-Draft Ribose
Intended status: Informational W. Wong
Expires: June 18, 2018 Hang Seng Management College
December 15, 2017

The SM4 Blockcipher Algorithm And Its Modes Of Operations
draft-ribose-cfrg-sm4-08

Abstract

This document describes the SM4 symmetric blockcipher algorithm published as GB/T 32907-2016 by the State Cryptography Administration of China (SCA).

This document is a product of the Crypto Forum Research Group (CFRG).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

SM4 [GBT.32907-2016] [ISO.IEC.18033-3.AMD2] is a cryptographic standard issued by the State Cryptography Administration (SCA) of China [SCA] (formerly the Office of State Commercial Cryptography Administration, OSCCA) as an authorized cryptographic algorithm for the use within China. The algorithm is published in public.

SM4 is a symmetric encryption algorithm, specifically a blockcipher, designed for data encryption.

1.1. Purpose

This document does not aim to introduce a new algorithm, but to provide a clear and open description of the SM4 algorithm in English, and also to serve as a stable reference for IETF documents that utilize this algorithm.

While this document is similar to [SM4-En] in nature, [SM4-En] is a textual translation of the "SMS4" algorithm [SM4] published in 2006. Instead, this document follows the updated description and structure of [GBT.32907-2016] published in 2016. Sections 1 to 7 of this document directly map to the corresponding sections numbers of the [GBT.32907-2016] standard for convenience of the reader.

This document also provides additional information on the design considerations of the SM4 algorithm [SM4-Details], its modes of operations that are currently being used (see Section 8), and the offical SM4 OIDs (see Section 9).

1.2. History

The "SMS4" algorithm (the former name of SM4) was invented by Shu-Wang Lu [LSW-Bio]. It was first published in 2003 as part of [GB.15629.11-2003], then published independently in 2006 by SCA (OSCCA at that time) [SM4], published as an industry cryptographic standard and renamed to "SM4" in 2012 by SCA (OSCCA at that time) [GMT-0002-2012], and finally formalized in 2016 as a Chinese National

Standard (GB Standard) [GBT.32907-2016]. SM4 has also been standardized in [ISO.IEC.18033-3.AMD2] by the International Organization for Standardization in 2017.

SMS4 was originally created for use in protecting wireless networks [SM4], and is mandated in the Chinese National Standard for Wireless LAN WAPI (Wired Authentication and Privacy Infrastructure) [GB.15629.11-2003]. A proposal was made to adopt SMS4 into the IEEE 802.11i standard, but the algorithm was eventually not included due to concerns of introducing inoperability with existing ciphers.

The latest SM4 standard [GBT.32907-2016] was proposed by the SCA (OSCCA at that time), standardized through TC 260 of the Standardization Administration of the People's Republic of China (SAC), and was drafted by the following individuals at the Data Assurance and Communication Security Research Center (DAS Center) of the Chinese Academy of Sciences, the China Commercial Cryptography Testing Center and the Beijing Academy of Information Science & Technology (BAIST):

o Shu-Wang Lu

o Dai-Wai Li

o Kai-Yong Deng

o Chao Zhang

o Peng Luo

o Zhong Zhang

o Fang Dong

o Ying-Ying Mao

o Zhen-Hua Liu

2. Terms and Definitions

The key words "*MUST*", "*MUST NOT*", "*REQUIRED*", "*SHALL*", "*SHALL NOT*", "*SHOULD*", "*SHOULD NOT*", "*RECOMMENDED*", "*MAY*", and "*OPTIONAL*" in this document are to be interpreted as described in [RFC2119].

The following terms and definitions apply to this document.

block length

Bit-length of a message block.

key length
Bit-length of a key.

key expansion algorithm
An operation that converts a key into a round key.

rounds
The number of iterations that the round function is run.

round key
A key used in each round on the blockcipher, derived from the input key, also called a subkey.

word
a 32-bit quantity

S-box
The S (substitution) box function produces 8-bit output from 8-bit input, represented as S(.)

3. Symbols And Abbreviations

S xor T
bitwise exclusive-or of two 32-bit vectors S and T. S and T will always have the same length.

a <<< i
32-bit bitwise cyclic shift on a with i bits shifted left.

4. Compute Structure

The SM4 algorithm is a blockcipher, with block size of 128 bits and a key length of 128 bits.

Both encryption and key expansion use 32 rounds of a nonlinear key schedule per block. Each round processes one of the four 32-bit words that constitute the block.

The structure of encryption and decryption are identical, except that the round key schedule has its order reversed during decryption.

Using a 8-bit S-box, it only uses exclusive-or, cyclic bit shifts and S-box lookups to execute.

5. Key And Key Parameters

The SM4 encryption key is 128 bits long and represented below, where each MK_i, (i = 0, 1, 2, 3) is 32 bits long.

MK = (MK_0, MK_1, MK_2, MK_3)

The round key schedule is derived from the encryption key, represented as below where each rk_i (i = 0, ..., 31) is 32 bits long:

(rk_0, rk_1, ... , rk_31)

The family key used for key expansion is represented as FK, where each FK_i (i = 0, ..., 3) is 32 bits long:

FK = (FK_0, FK_1, FK_2, FK_3)

The constant key used for key expansion is represented as CK, where each CK_i (i = 0, ..., 31) is 32 bits long:

CK = (CK_0, CK_1, ... , CK_31)

6. Functions

6.1. Round Function F

The round function F is defined as:

F(X_0, X_1, X_2, X_3, rk) = X_0 xor T(X_1 xor X_2 xor X_3 xor rk)

Where:

o Each $$X_i$ is 32-bit wide.

o The round key rk is 32-bit wide.

6.2. Permutations T and T'

T is a reversible permutation that outputs 32 bits from a 32-bit input.

It consists of a nonlinear transform tau and linear transform L.

T(.) = L(tau(.))

The permutation T' is created from T by replacing the linear transform function L with L'.

T'(.) = L'(tau(.))

6.2.1. Nonlinear Transformation tau

tau is composed of four parallel S-boxes.

Given a 32-bit input A, where each a_i is a 8-bit string:

A = (a_0, a_1, a_2, a_3)

The output is a 32-bit B, where each b_i is a 8-bit string:

B = (b_0, b_1, b_2, b_3)

B is calculated as follows:

(b_0, b_1, b_2, b_3) = tau(A)

tau(A) = (S(a_0), S(a_1), S(a_2), S(a_3))

6.2.2. Linear Transformations L and L'

The output of nonlinear transformation function tau is used as input to linear transformation function L.

Given B, a 32-bit input.

The linear transformation L' is defined as follows.

L(B) = B xor (B <<< 2) xor (B <<< 10) xor (B <<< 18) xor (B <<< 24)

The linear transformation L' is defined as follows.

L'(B) = B xor (B <<< 13) xor (B <<< 23)

6.2.3. S-box S

The S-box S used in nonlinear transformation tau is given in the lookup table shown in Figure 1 with hexadecimal values.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	D6	90	E9	FE	CC	E1	3D	B7	16	B6	14	C2	28	FB	2C	05
1	2B	67	9A	76	2A	BE	04	C3	AA	44	13	26	49	86	06	99
2	9C	42	50	F4	91	EF	98	7A	33	54	0B	43	ED	CF	AC	62
3	E4	B3	1C	A9	C9	08	E8	95	80	DF	94	FA	75	8F	3F	A6
4	47	07	A7	FC	F3	73	17	BA	83	59	3C	19	E6	85	4F	A8
5	68	6B	81	B2	71	64	DA	8B	F8	EB	0F	4B	70	56	9D	35
6	1E	24	0E	5E	63	58	D1	A2	25	22	7C	3B	01	21	78	87
7	D4	00	46	57	9F	D3	27	52	4C	36	02	E7	A0	C4	C8	9E
8	EA	BF	8A	D2	40	C7	38	B5	A3	F7	F2	CE	F9	61	15	A1
9	E0	AE	5D	A4	9B	34	1A	55	AD	93	32	30	F5	8C	B1	E3
A	1D	F6	E2	2E	82	66	CA	60	C0	29	23	AB	0D	53	4E	6F
B	D5	DB	37	45	DE	FD	8E	2F	03	FF	6A	72	6D	6C	5B	51
C	8D	1B	AF	92	BB	DD	BC	7F	11	D9	5C	41	1F	10	5A	D8
D	0A	C1	31	88	A5	CD	7B	BD	2D	74	D0	12	B8	E5	B4	B0
E	89	69	97	4A	0C	96	77	7E	65	B9	F1	09	C5	6E	C6	84
F	18	F0	7D	EC	3A	DC	4D	20	79	EE	5F	3E	D7	CB	39	48

Figure 1: SM4 S-box Values

For example, input "EF" will produce an output read from the S-box table row E and column F, giving the result S(EF) = 84.

7. Algorithm

7.1. Encryption

The encryption algorithm consists of 32 rounds and 1 reverse transform R.

Given a 128-bit plaintext input, where each X_i is 32-bit wide:

(X_0, X_1, X_2, X_3)

The output is a 128-bit ciphertext, where each Y_i is 32-bit wide:

(Y_0, Y_1, Y_2, Y_3)

Each round key is designated as rk_i, where each rk_i is 32-bit wide and i = 0, 1, 2, ..., 31.

a. 32 rounds of calculation

i = 0, 1, ..., 31

X_{i+4} = F(X_i, X_{i+1}, X_{i+2}, X_{i+3}, rk_i)

b. reverse transformation

$(Y_0, Y_1, Y_2, Y_3) = R(X_{32}, X_{33}, X_{34}, X_{35})$

$R(X_{32}, X_{33}, X_{34}, X_{35}) = (X_{35}, X_{34}, X_{33}, X_{32})$

Please refer to Appendix A for sample calculations.

A flow of the calculation is given in Figure 2.

```
                              128-bits plaintext
                         \___________________________/
                                       |
                                       v
                          X_0     X_1     X_2     X_3
                           |       |       |       |
                           v       v       v       v
                         +---------------------------+
             Round 1     |      Round Function       | <--rk_0
                         +---------------------------+
                           |       |       |       |
                          X_1     X_2     X_3     X_4
                           |       |       |       |
                           v       v       v       v
                         +---------------------------+
             Round 2     |      Round Function       | <--rk_1
                         +---------------------------+
                           |       |       |       |
                          X_2     X_3     X_4     X_5
                           |       |       |       |
                           v       v       v       v
                                      ...

                          X_31    X_32    X_33    X_34
                           |       |       |       |
                           v       v       v       v
                         +---------------------------+
             Round 32    |      Round Function       | <--rk_31
                         +---------------------------+
                           |       |       |       |
                          X_32    X_33    X_34    X_35
                           |       |       |       |
                           v       v       v       v
                         +---------------------------+
                         | Reverse Transformation R  |
                         +---------------------------+
                           |       |       |       |
                          Y_0     Y_1     Y_2     Y_3

                         \___________________________/
                                       |
                                       v
                              128-bits ciphertext
```

Figure 2: SM4 Encryption Flow

7.2. Decryption

Decryption takes an identical process as encryption, with the only difference the order of the round key sequence.

During decryption, the round key sequence is:

```
(rk_31, rk_30, ..., rk_0)
```

7.3. Key Schedule

Round keys used during encryption are derived from the encryption key.

Specifically, given the encryption key MK, where each MK_i is 32-bit wide:

```
MK = (MK_0, MK_1, MK_2, MK_3)
```

Each round key rk_i is created as follows, where i = 0, 1, ..., 31.

```
(K_0, K_1, K_2, K_3) =
    (MK_0 xor FK_0, MK_1 xor FK_1, MK_2 xor FK_2, MK_3 xor FK_3)

rk_i = K_{i + 4}

K_{i + 4} =
    K_i xor T' (K_{i + 1} xor K_{i + 2} xor K_{i + 3} xor CK_i)
```

Since the decryption key is identical to the encryption key, the round keys used in the decryption process are derived from the decryption key through the identical process to that of during encryption.

Figure 3 depicts the i-th round of SM4.

```
               X_i               rk_i  X_{i+1} X_{i+2} X_{i+3}
                |                 |      |       |       |
                |                 |      |       |       |
                |                 v      |       |       |
              +---+    +---+    +---+    |       |       |
              | X |    |   |    | X | <--+       |       |
              | O | <- | T | <- | O | <----------+       |
              | R |    |   |    | R | <------------------+
              +---+    +---+    +---+    |       |       |
                |                       /       /       /
                |                      /       /       /
                |                     /       /       /
                 \--------------------------------------=
                                    /       /       /    \
                                   /       /       /     |
                  /---------------/       /       /      |
                 |                       |       |       |
              X_{i+1}                 X_{i+2} X_{i+3} X_{i+4}
```

Figure 3: SM4 Round Function For the i-th Round

7.3.1. Family Key FK

Family key FK given in hexadecimal notation, is:

```
FK_0 = A3B1BAC6
FK_1 = 56AA3350
FK_2 = 677D9197
FK_3 = B27022DC
```

7.3.2. Constant Key CK

The method to retrieve values from the constant key CK is as follows.

Let ck_{i, j} be the j-th byte (i = 0, 1, ..., 31; j = 0, 1, 2, 3) of CK_i.

Therefore, each ck_{i, j} is a 8-bit string, and each CK_i a 32-bit word.

CK_i = (ck_{i, 0}, ck_{i, 1}, ck_{i, 2}, ck_{i, 3})

ck_{i, j} = (4i + j) x 7 (mod 256)

The values of the constant key CK_i, where (i = 0, 1, ..., 31), in hexadecimal, are:

CK_0	=	00070E15	CK_16	=	C0C7CED5
CK_1	=	1C232A31	CK_17	=	DCE3EAF1
CK_2	=	383F464D	CK_18	=	F8FF060D
CK_3	=	545B6269	CK_19	=	141B2229
CK_4	=	70777E85	CK_20	=	30373E45
CK_5	=	8C939AA1	CK_21	=	4C535A61
CK_6	=	A8AFB6BD	CK_22	=	686F767D
CK_7	=	C4CBD2D9	CK_23	=	848B9299
CK_8	=	E0E7EEF5	CK_24	=	A0A7AEB5
CK_9	=	FC030A11	CK_25	=	BCC3CAD1
CK_10	=	181F262D	CK_26	=	D8DFE6ED
CK_11	=	343B4249	CK_27	=	F4FB0209
CK_12	=	50575E65	CK_28	=	10171E25
CK_13	=	6C737A81	CK_29	=	2C333A41
CK_14	=	888F969D	CK_30	=	484F565D
CK_15	=	A4ABB2B9	CK_31	=	646B7279

8. Modes of Operation

This document defines multiple modes of operation for the SM4 blockcipher algorithm.

The CBC (Cipher Block Chaining), ECB (Electronic CodeBook), CFB (Cipher FeedBack), OFB (Output FeedBack) and CTR (Counter) modes are defined in [NIST.SP.800-38A] and utilized with the SM4 algorithm in the following sections.

8.1. Variables And Primitives

Hereinafter we define:

SM4Encrypt(P, K)
   The SM4 algorithm that encrypts plaintext P with key K, described in Section 7.1

SM4Decrypt(C, K)
   The SM4 algorithm that decrypts ciphertext C with key K, described in Section 7.2

b
   block size in bits, defined as 128 for SM4

P_j
   block j of ciphertext bitstring P

C_j
   block j of ciphertext bitstring C

NBlocks(B, b)
   Number of blocks of size b-bit in bitstring B

IV
   Initialization vector

LSB(b, S)
   Least significant b bits of the bitstring S

MSB(b, S)
   Most significant b bits of the bitstring S

8.2. Initialization Vectors

The CBC, CFB and OFB modes require an additional input to the encryption process, called the initialization vector (IV). The identical IV is used in the input of encryption as well as the decryption of the corresponding ciphertext.

Generation of IV values *MUST* take into account of the considerations in Section 12 recommended by [BC-EVAL].

8.3. SM4-ECB

In SM4-ECB, the same key is utilized to create a fixed assignment for a plaintext block with a ciphertext block, meaning that a given plaintext block always gets encrypted to the same ciphertext block. As described in [NIST.SP.800-38A], this mode should be avoided if this property is undesirable.

This mode requires input plaintext to be a multiple of the block size, which in this case of SM4 it is 128-bit. It also allows multiple blocks to be computed in parallel.

8.3.1. SM4-ECB Encryption

Inputs:

o P, plaintext, length *MUST* be multiple of b

o K, SM4 128-bit encryption key

Output:

o C, ciphertext, length is a multiple of b

C is defined as follows.

```
n = NBlocks(P, b)

for i = 1 to n
  C_i = SM4Encrypt(P_i, K)
end for

C = C_1 || ... || C_n
```

8.3.2. SM4-ECB Decryption

Inputs:

- o C, ciphertext, length *MUST* be multiple of b

- o K, SM4 128-bit encryption key

Output:

- o P, plaintext, length is a multiple of b

P is defined as follows.

```
n = NBlocks(C, b)

for i = 1 to n
  P_i = SM4Decrypt(C_i, K)
end for

P = P_1 || ... || P_n
```

8.4. SM4-CBC

SM4-CBC is similar to SM4-ECB that the input plaintext *MUST* be a multiple of the block size, which is 128-bit in SM4. SM4-CBC requires an additional input, the IV, that is unpredictable for a particular execution of the encryption process.

Since CBC encryption relies on a forward cipher operation that depend on results of the previous operation, it cannot be parallelized. However, for decryption, since ciphertext blocks are already available, CBC parallel decryption is possible.

8.4.1. SM4-CBC Encryption

Inputs:

- P, plaintext, length *MUST* be multiple of b
- K, SM4 128-bit encryption key
- IV, 128-bit, unpredictable, initialization vector

Output:

- C, ciphertext, length is a multiple of b

C is defined as follows.

```
n = NBlocks(P, b)

C_1 = SM4Encrypt(P_1 xor IV, K)

for i = 2 to n
  C_i = SM4Encrypt(P_i xor C_{i - 1}, K)
end for

C = C_1 || ... || C_n
```

8.4.2. SM4-CBC Decryption

Inputs:

- C, ciphertext, length *MUST* be a multiple of b
- K, SM4 128-bit encryption key
- IV, 128-bit, unpredictable, initialization vector

Output:

- P, plaintext, length is multiple of b

P is defined as follows.

```
n = NBlocks(C, b)

P_1 = SM4Decrypt(C_1, K) xor IV

for i = 2 to n
  P_i = SM4Decrypt(C_i, K) xor C_{i - 1}
end for

P = P_1 || ... || P_n
```

8.5. SM4-CFB

SM4-CFB relies on feedback provided by successive ciphertext segments to generate output blocks. The plaintext given must be a multiple of the block size.

Similar to SM4-CBC, SM4-CFB requires an IV that is unpredictable for a particular execution of the encryption process.

SM4-CFB further allows setting a positive integer parameter s, that is less than or equal to the block size, to specify the size of each data segment. The same segment size must be used in encryption and decryption.

In SM4-CFB, since the input block to each forward cipher function depends on the output of the previous block (except the first that depends on the IV), encryption is not parallelizable. Decryption, however, can be parallelized.

8.5.1. SM4-CFB Variants

SM4-CFB takes an integer s to determine segment size in its encryption and decryption routines. We define the following variants of SM4-CFB for various s:

- SM4-CFB-1, the 1-bit SM4-CFB mode, where s is set to 1.

- SM4-CFB-8, the 8-bit SM4-CFB mode, where s is set to 8.

- SM4-CFB-64, the 64-bit SM4-CFB mode, where s is set to 64.

- SM4-CFB-128, the 128-bit SM4-CFB mode, where s is set to 128.

8.5.2. SM4-CFB Encryption

Inputs:

- o P#, plaintext, length *MUST* be multiple of s
- o K, SM4 128-bit encryption key
- o IV, 128-bit, unpredictable, initialization vector
- o s, an integer 1 <= s <= b that defines segment size

Output:

- o C#, ciphertext, length is a multiple of s

C# is defined as follows.

```
n = NBlocks(P#, s)

I_1 = IV
for i = 2 to n
  I_i = LSB(b - s, I_{i - 1}) || C#_{j - 1}
end for

for i = 1 to n
  O_j = SM4Encrypt(I_i, K)
end for

for i = 1 to n
  C#_i = P#_1 xor MSB(s, O_j)
end for

C# = C#_1 || ... || C#_n
```

8.5.3. SM4-CFB Decryption

Inputs:

- o C#, ciphertext, length *MUST* be a multiple of s
- o K, SM4 128-bit encryption key
- o IV, 128-bit, unpredictable, initialization vector

o s, an integer 1 <== s <== b that defines segment size

Output:

o P#, plaintext, length is multiple of s

P# is defined as follows.

```
n = NBlocks(P#, s)

I_1 = IV
for i = 2 to n
  I_i = LSB(b - s, I_{i - 1}) || C#_{j - 1}
end for

for i = 1 to n
  O_j = SM4Encrypt(I_i, K)
end for

for i = 1 to n
  P#_i = C#_1 xor MSB(s, O_j)
end for

P# = P#_1 || ... || P#_n
```

8.6. SM4-OFB

SM4-OFB is the application of SM4 through the Output Feedback mode. This mode requires that the IV is a nonce, meaning that the IV *MUST* be unique for each execution for an input key. OFB does not require the input plaintext to be a multiple of the block size.

In OFB, the routines for encryption and decryption are identical. As each forward cipher function (except the first) depends on previous results, both routines cannot be parallelized. However given a known IV, output blocks could be generated prior to the input of plaintext (encryption) or ciphertext (decryption).

8.6.1. SM4-OFB Encryption

Inputs:

o P, plaintext, composed of (n - 1) blocks of size b, with the last block P_n of size 1 <== u <== b

o K, SM4 128-bit encryption key

o IV, a nonce (a unique value for each execution per given key)

Output:

o C, ciphertext, composed of (n - 1) blocks of size b, with the last block C_n of size 1 <== u <== b

C is defined as follows.

```
n = NBlocks(P, b)

I_1 = IV
for i = 1 to (n - 1)
  O_i = SM4Encrypt(I_i)
  I_{i + 1} = O_i
end for

for i = 1 to (n - 1)
  C_i = P_i xor O_i
end for

C_n = P_n xor MSB(u, O_n)

C = C_1 || ... || C_n
```

8.6.2. SM4-OFB Decryption

Inputs:

o C, ciphertext, composed of (n - 1) blocks of size b, with the last block C_n of size 1 <== u <== b

o K, SM4 128-bit encryption key

o IV, the nonce used during encryption

Output:

o P, plaintext, composed of (n - 1) blocks of size b, with the last block P_n of size 1 <== u <== b

C is defined as follows.

```
n = NBlocks(C, b)

I_1 = IV
for i = 1 to (n - 1)
  O_i = SM4Encrypt(I_i)
  I_{i + 1} = O_i
end for

for i = 1 to (n - 1)
  P_i = C_i xor O_i
end for

P_n = C_n xor MSB(u, O_n)

P = P_1 || ... || P_n
```

8.7. SM4-CTR

SM4-CTR is an implementation of a stream cipher through a blockcipher primitive. It generates a "keystream" of keys that are used to encrypt successive blocks, with the keystream created from the input key, a nonce (the IV) and an incremental counter. The counter could be any sequence that does not repeat within the block size.

Both SM4-CTR encryption and decryption routines could be parallelized, and random access is also possible.

8.7.1. SM4-CTR Encryption

Inputs:

o P, plaintext, composed of (n - 1) blocks of size b, with the last block P_n of size 1 <== u <== b

o K, SM4 128-bit encryption key

o IV, a nonce (a unique value for each execution per given key)

o T, a sequence of counters from T_1 to T_n

Output:

o C, ciphertext, composed of (n - 1) blocks of size b, with the last block C_n of size 1 <== u <== b

C is defined as follows.

```
n = NBlocks(P, b)

for i = 1 to n
  O_i = SM4Encrypt(T_i)
end for

for i = 1 to (n - 1)
  C_i = P_i xor O_i
end for

C_n = P_n xor MSB(u, O_n)

C = C_1 || ... || C_n
```

8.7.2. SM4-CTR Decryption

Inputs:

- o C, ciphertext, composed of (n - 1) blocks of size b, with the last block C_n of size 1 <= u <= b
- o K, SM4 128-bit encryption key
- o IV, a nonce (a unique value for each execution per given key)
- o T, a sequence of counters from T_1 to T_n

Output:

- o P, plaintext, composed of (n - 1) blocks of size b, with the last block P_n of size 1 <= u <= b

P is defined as follows.

```
n = NBlocks(C, b)

for i = 1 to n
  O_i = SM4Encrypt(T_i)
end for

for i = 1 to (n - 1)
  P_i = C_i xor O_i
end for

P_n = C_n xor MSB(u, O_n)

C = C_1 || ... || C_n
```

9. Object Identifier

The Object Identifier for SM4 is identified through these OIDs.

9.1. GM/T OID

"1.2.156.10197.1.104" for "SM4 Algorithm" [GMT-0006-2012].

9.2. ISO OID

"1.0.18033.3.2.4" for "id-bc128-sm4" [ISO.IEC.18033-3.AMD2], described below.

- "is18033-3" {iso(1) standard(0) is18033(18033) part3(3)}

- "id-bc128" {is18033-3 block-cipher-128-bit(2)}

- "id-bc128-sm4" {id-bc128 sm4(4)}

10. Design Considerations

10.1. Basic Transformation

The chaos principle and the diffusion principle are two basic principles of block cipher design. A well-designed blockcipher algorithm should be based on a cryptographically sound basic transformation structure, with its round calculation based on a cryptographically sound basic transformation.

The cryptographic properties of the basic transformation determines the efficiency of the resulting encryption transformation.

The SM4 algorithm is structured on orthomorphic permutation. Its round transformation is an orthomorphic permutation, and its cryptographic properties can be deduced from the characteristics of orthomorphic permutations.

Let the single round of the SM4 block cipher algorithm be P, for any given plaintext X, P (X, K ')! = P (X, K) if the key K'! = K.

The conclusion shows that if X is a row variable and K is a column variable, the square P(X, K) forms a Latin square. There are two conclusions about the nature of cryptography:

1. The SM4 blockcipher algorithm will produce different round transformations given different keys.

2. The SM4 blockcipher algorithm, within a single round, will produce a different output given the same input with different keys.

10.2. Nonlinear Transformation

An S-box can be viewed as a bijection:

S(X) = (f_1(X), f_2(X), ... , f_m(X)) : F_2^n -> F_2^m.

S(x): F_2^n -> F_2^m can be represented as a multi-output boolean function with n-bit input and m-bit output, or a n x m S-box (an S-box with n inputs and m outputs), usually realized as a substitution that takes an n-bit input and produces a m-bit output. In SM4, the S-box takes n = m = 8.

In many blockciphers, the S-box is the sole element providing nonlinearity, for the purpose of mixing, in order to reduce linearity and to hide its variable structure.

The cryptographic properties of the S-box directly affects the resulting cryptographic strength of the blockcipher. When designing a blockcipher, the cryptographic strength of the S-box must be taken into account. The cryptographic strength of an S-box can be generally measured by factors such as its nonlinearity and differential distribution.

10.2.1. S-box Algebraic Expression

In order to prevent insertion attacks, the algebraic formula used for cryptographic substitution should be a high degree polynomial and contain a large number of terms.

The algebraic expression of the SM4 S-box [SM4-Sbox] is determined through Lagrange's interpolation to be a polynomial of the 254th degree with 255 terms, providing the highest level of complexity based on its size:

```
f(x) : sum_{i=0}^{255} y_i
                PI_{j!=i, j=0}^255 ((x - x_j) / (x_i - x_j))
```

10.2.2. Algebraic Degree And Distribution Of Terms

Any n boolean function f(x): F_2^n -> F_2 can be represented uniquely in its algebraic normal form shown below:

```
f(X) = a_0 + sum_{1<=i_i<...<i_k<=n, 1<=k<=n}
                      a_{i_1 i_2 ... i_k} x_{i_1} x_{i_2} ... x_{i_k}

X = (x_1, x_2, ..., x_n)

a_0, a_{i_1, i_2, ... i_k} element-of F_2
```

The "algebraic degree" of the n-boolean function f(X) is defined to be the algebraic degree of the highest algebraic degree of its terms with a nonzero coefficient in its ANF representation. The constant of the i-th term of f(x) in ANF representation is called the i-th term of f(X), the total number of all i-th (0<=i<=n) terms is called the "number of terms" of f(X).

S(X) can be represented as a m-component function S(X) = (f_1(X), f_2(X), ... f_m(X)): F_2^n -> F_2^m. Consider S(X) to be a random substitution, each of its component functions would be best to have algebraic degree of n-1, each component function i-th coefficient should be near C_n^i/2. If the algebraic degree is too low, for example, each component function has a degree of 2, then the algorithm can be easily attacked by advanced differential cryptanalysis. If the number of terms are insufficient, then it may improve the success probability of insert attacks.

The algebraic degrees and number of terms of the SM4 S-box are described in Figure 4.

Component Function	Algebraic Degree								
	8	7	6	5	4	3	2	1	0
Y_0	0	3	15	31	28	29	14	3	1
Y_1	0	3	12	34	40	33	12	4	1
Y_2	0	5	17	24	40	24	11	3	0
Y_3	0	2	11	31	34	27	15	5	1
Y_4	0	5	15	28	33	24	13	5	0
Y_5	0	5	11	25	41	25	16	4	1
Y_6	0	4	15	29	27	32	18	4	1
Y_7	0	4	14	32	35	30	16	3	0
Expected Value	1/2	4	14	28	35	28	14	4	1/2

Figure 4: SM4 S-box Component Functions Algebraic Degree And Terms

10.2.3. Differential Distribution

The definition of differential distribution has been given in [BC-Design].

Differential cryptanalysis is a chosen-plaintext attack, with the understanding that analysis of selected plaintexts of differentials can retrive the most probable key. Differential distribution is an attribute to measure the resistance of a cryptographic function against differential cryptanalysis.

```
delta_S = 1/2^n max_{a in F_2^n, a!=0} max_{beta in F_2^m} |
  { X in F_2^n : S(X and alpha) - S(X) = beta } |
```

"delta_S" is the differential distribution of the S-box "S".

According to the definition of differential distribution, 2^{-m} <= delta_S <= 2^{m-n}, if there is a delta_S = 2^{m-n} then S is considered a fully nonlinear function from F_2^n to F_2^m. For resistance against differential cryptanalysis, the differential distribution should be as low as possible.

The highest differential distribution of the SM4 S-box is 2^{-6}, meaning it has a good resistance against differential cryptanalysis.

10.2.4. Nonlinearity

The nonlinearity of an S-box is described by [BC-Design].

Let $S(X) = (f_1(X), f_2(X), \ldots, f_m(X)) : F_2^n \rightarrow F_2^m$ be a multi-output function. The nonlinearity of $S(X)$ is defined as $N_S = \min_{l \in L_n, 0 \neq u \in F_2^m} d_H(u \cdot S(X), l(X))$.

L_n is the group of all n-boolean functions, $d_H(f, l)$ is the Hamming distance between f and l. The nonlinearity of the S-box is in fact the minimum Hamming distance between all the Boolean functions and all affine functions.

The upper-bound of nonlinearity is known to be $2^{n-1} - 2^{n/2 - 1}$, where a Boolean function that reaches this bound is called a "bent function".

The nonlinearity of a Boolean function is used to measure resistance against linear attacks. The higher the nonlinearity, the higher resistance that the Boolean function f(x) has against linear attacks. On the contrary, the lower the nonlinearity, the Boolean function f(x) has lower resistance against linear attacks.

The nonlinearity of the SM4 S-box is 112.

10.2.5. Maximum Linearity Advantage

Linear approximation of a S-box is defined in [BC-Design]. Given a S-box with n inputs and m outputs, any linear approximation can be represented as : $a \cdot X = b \cdot Y$, where $a \in F_2^n$, $b \in F_2^m$.

The probability p that satisfies $a \cdot X = b \cdot Y$ is

$$| p - 1/2 | \leq 1/2 - N_S / 2^n$$

where $| p - 1/2 |$ is the advantage of the linear approximation equation, $\lambda_S = 1/2 - N_s / 2^n$ is the maximum advantage of the S-box.

The maximum advantage of the SM4 S-box is 2^{-4}.

10.2.6. Balance

A S-box $S(X) = (f_1(X), f_2(X), \ldots, f_m(X)) : F_2^n \rightarrow F_2^m$ is considered "balanced" if for any beta in F_2^m, there are 2^{n-m} x in F_2^n, such that $S(x) = \beta$.

The SM4 S-box is balanced.

10.2.7. Completness and Avalanche Effect

A S-box S(X) = (f_1(X), f_2(X), ... , f_m(X)) : F_2^n -> F_2^m is considered "complete" if every input bit directly correlates to an output bit.

In algebraic expression, each component function contains the unknown variables x_1, x_2, ... x_n, such that for any (s, t) in { (i, j) | 1 <= i <= n, 1 <= j <= m}, there is an X that S(X) and S(X and e_s) would contain a different bit t.

Avalanche effect refers to a single bit change in the input would correspond to a change of half of the output bits.

The SM4 S-box satisfies completness and the avalanche effect.

10.3. Linear Transform

Linear transformation is used to provide diffusion in SM4. A blockcipher algorithm often adopts m x m S-boxes to form an obfuscation layer.

Since the m-bits output by one S-box are only related to the m bits of its input and are irrelevant to the input of other S boxes, the introduction of a linear transform would disrupt and mix the output m-bits so that they seem correlating to the other S-box inputs.

A sound linear transform design will diffuse the S-box output, allowing the blockcipher to resist differential and linear cryptanalysis.

An important measure of the diffusivity of a linear transform is its branch number.

The "branch number" of a linear transform is defined in [BC-Design]:

B(theta) = min_{x!=0} w_b(x) + w_b(theta(x))

Where B(theta) is the branch number of transform theta, w_b(x) is a non-zero integer x_i (1 <== i <== m), and x_i is called the "bundle weight".

The branch number can be used to quantify the resistance of the block cipher algorithm to differential cryptanalysis and linear cryptanalysis.

Similar to differential cryptanalysis and linear cryptanalysis, the differential branch number and linear branch number of theta can be defined as follows.

The differential branch number of theta is:

```
B_d(theta) = min_{x, x!= x*}
               (w_b(x and x*) + w_b(theta(x)) and theta(x*))
```

The linear branch number of theta is:

```
B_l(theta) = min_{a, b, c (x . alpha^t , theta(x) . beta) != 0}
               (w_b(alpha) + w_b(beta))

  where,
    c (x . a^t , theta(x) . beta) =
                      2 X Pr(x . alpha^t = theta(x) . beta) - 1
    x . alpha^t  is a matrix multiplication.
```

The branch number in a linear transformation reflects its diffusivity. The higher the branch number, the better the diffusion effect.

This means that the larger the differential branch number or linear branch number, the more known plaintexts will be required for differential or linear cryptanalysis respectively.

The linear transform differential branch number and linear branch number of SM4 are both 5.

10.4. Key Expansion Algorithm

The SM4 key schedule is designed to fulfill the security requirements of the encryption algorithm and achieve ease of implementation for performance reasons.

All subkeys are derived from the encryption key, and therefore, subkeys are always statistically relevant. In the context of a blockcipher, it is not possible to have non-statistical-correlated subkeys, but the designer can only aim to have subkeys achieve near statistical independence [BC-Design].

The purpose of the key schedule, generated through the key expansion algorithm, is to mask the statistical correlation between subkeys to make this relationship difficult to exploit.

The SM4 key expansion algorithm satisfies the following design criteria:

1. There are no obvious statistical correlation between subkeys;

2. There are no weak subkeys;

3. The speed of key expansion is not slower than the encryption algorithm, and uses less resources;

4. Every subkey can be directly generated from the encryption key.

11. Cryptanalysis Results

SM4 has been heavily cryptanalyzed by international researchers since it was first published. Nearly all currently known cryptanalysis techniques have been applied to SM4.

At the time of publishing this document, there are no known practical attacks against the full SM4 blockcipher. However, there are side-channel concerns [SideChannel] when the algorithm is implemented in a hardware device.

A summary of cryptanalysis results are presented in the following sections.

11.1. Differential Cryptanalysis

In 2008, Zhang et al. [SM4-DiffZhang1] gave a 21-round differential analysis with data complexity 2^188, time complexity 2^126.8 encryptions.

In 2008, Kim et al. [SM4-LDA] gave a 22-round differential attack that requires 2^118 chosen plaintexts, 2^123 memory and 2^125.71 encryptions.

In 2009, Zhang et al. (differing author but overlapping team) [SM4-DiffZhang2] gave a 18-round differential characteristics with an attack that reaches the 22nd round, with data complexity 2^117 and time complexity 2^112.3.

In 2010, Zhang et al. (with no relation to above) [SM4-DiffZhang3] utilized 18-round differential characteristics for the 22nd round with 2^117 chosen plaintexts with time complexity 2^123 encryptions, memory complexity of 2^112.

In 2011, Su et al. [SM4-DiffSu] gave a 19 round differential characteristics and pushed their attack to the 23rd round, with data complexity of 2^118 chosen plaintexts, time complexity 2^126.7 encryptions, and memory complexity 2^120.7.

11.2. Linear Cryptanalysis

In 2008 Etrog et al. [SM4-LinearEtrog] provided a linear cryptanalysis result for 22 rounds of SM4, the data complexity is given as 2^188.4 known plaintexts, time complexity 2^117 encrypt operations.

In the same year, Kim et al. [SM4-LDA] improved on the linear cryptanalysis result for 22 rounds of SM4 with data complexity of 2^117 known plaintexts, memory complexity of 2^109 and time complexity of 2^109.86.

In 2011 Dong [SM4-LinearDong] presented a linear cryptanalysis result for 20 rounds, 2^110.4 known ciphertexts, 2^106.8 encryption operations, memory complexity 2^90.

In 2014 Liu et al. [SM4-LinearLiu] presented their linear cryptanalysis for 23-rounds of SM4, time complexity 2^112 encryption operations, data complexity 2^126.54 known ciphertexts, memory complexity 2^116.

In 2017 Liu et al. [SM4-NLC] presented an attack based on linear cryptanalysis on 24-rounds of SM4, with time complexity of 2^122.6 encryptions, data complexity of 2^122.6 known ciphertexts, and memory complexity of 2^85.

11.3. Multi-dimensional Linear Cryptanalysis

In 2010, Liu et al. [SM4-MLLiu] constructed a series of 18 rounds of linear traces based on a 5-round circular linear trace, capable of attacking 22 rounds of SM4. The required data complexity was 2^112 known plaintexts, time complexity 2^124.21 encryption operations, with memory complexity of 2^118.83.

In 2010 Cho et al. [SM4-MLCho] gave a linear analysis of 23 rounds of SM4 with a data complexity of 2^126.7 known plaintexts and a time complexity of 2^127, memory complexity of 2^120.7.

In 2014, Liu et al. [SM4-LinearLiu] gave the results of multi-dimensional linear analysis of 23 rounds of SM4 algorithm. The time complexity was 2^122.7, data complexity was 2^122.6 known plaintext with memory complexity 2^120.6.

11.4. Impossible Differential Cryptanalysis

In 2007 Lu et al. [SM4-IDCLu] first presented 16 rounds of impossible differential analysis of SM4 with the required data

complexity 2^105 chosen plaintexts, time complexity 2^107 encryption operations.

In 2008 Toz et al. [SM4-IDCToz] revised the results of [SM4-IDCLu], that the data complexity is actually 2^117.05 chosen plaintexts, time complexity 2^132.06 encryptions, but its complexity is already beyond the 2^128 limit.

In 2010 Wang et al. [SM4-IDCWang] pushed the impossible differential cryptanalysis to 17 rounds of SM4, the data complexity is 2^117 chosen ciphertexts, time complexity 2^132 memory queries.

11.5. Zero-correlation Linear Cryptanalysis

In 2015 Ma et al. [SM4-ZCLC] gives the results of multi-dimensional zero-correlation linear cryptanalysis of a 14-round SM4 algorithm. The required data complexity is 2^123.5 known plaintexts, time complexity is 2^120.7 encryption operations and memory complexity of 2^73 blocks.

11.6. Integral Cryptanalysis

In 2007 Liu et al. [SM4-ICLiu] first gave a 13-round integral analysis of SM4, which required 2^16 chosen plaintexts and time complexity of 2^114 encryption operations.

In 2008 Zhong et al. [SM4-ICZhong] constructed a 12-round distinguisher of SM4 to attack 14-round SM4, with data complexity of 2^32 chosen plaintexts and time complexity 2^96.5 encryptions.

11.7. Algebraic Attacks

In 2009 Ji et al. [SM4-AAJi] and in 2010 Erickson et al. [SM4-AAEr] utilized algebraic methods such as XL, Groebner base and SAT to analyze the resistance of SM4 against algebraic attacks. The results demonstrate that SM4 is safe against algebraic attacks, and specifically, has a higher resistance against algebraic attacks than AES.

11.8. Matrix Attacks

In 2007 Lu et al. [SM4-IDCLu] provided a matrix attack against 14-round SM4, with data complexity 2^121.82 chosen plaintexts, time complexity 2^116.66 encryptions.

In 2008 Toz et al. [SM4-IDCToz] lowered both data and time complexity of the aforementioned attack to 2^106.89 chosen ciphertexts and time complexity of 2^107.89.

In 2008, Zhang et al. [SM4-DiffZhang1] provided a matrix attack against 16-round SM4, which required a data complexity of 2^125 chosen plaintexts and time complexity of 2^116 encryptions.

She's Master dissertation [SM4-MatrixShe] provided a SM4 16-round matrix distinguisher that can attack 18-round SM4, with data complexity of 2^127 chosen plaintexts and time complexity 2^110.77 encryptions with memory complexity of 2^130.

In 2012 Wei et al. [SM4-MatrixWei] applied differential analysis and algebraic attack techniques on 20-round SM4 and discovered that the combined attack results on 20-round SM4 are superior than using pure differential cryptanalysis.

11.9. Provable Security Against Differential And Linear Cryptanalysis

SM4 uses a novel structure differing from the general Feistel and SP structures.

[SM4-Random] has proven that the SM4 non-balanced Feistel structure is pseudo-random.

[SM4-SLDC] analyzes the SM4 non-balanced Feistel structure on its resistance against differential and linear cryptanalysis techniques. Under SP type round functions with branch number 5, it is proven that in a 27-round SM4 guarantees at least 22 active S-boxes, therefore SM4 is secure against differential attacks.

[SM4-SLC] has analyzed resistance of SM4 against linear cryptanalysis.

11.10. Provable Security Against Related-Key Differential Cryptanalysis

Related-key differential cryptanalysis is related to the encryption algorithm and key schedule. When performing a related-key attack, the attacker simultaneously insert differences in both the key and the message.

In [AutoDC], Sun et al. proposed an automated differential route search method based on MILP (mixed-integer linear programming) that can be used to assess the security bounds of a blockcipher under (related-key) differential cryptanalysis.

[SM4-RKDC] describes the lower bounds of active S-boxes within SM4 and is shown in Table 1.

Round	Single Key	Related Key
3	0	0
4	1	1
5	2	2
6	2	4
7	5	6
8	6	8
9	7	9
10	8	10
11	9	11
12	10	13
13	10	14
14	10	14
15	13	16
16	14	18
17	15	19
18	16	20
19	18	22
20	18	-
21	19	-
22	20	-
23	22	-
24	23	-
25	23	-
26	24	-

Table 1: Strongest SM4 Attacks ("-" denotes unknown)

As the maximal probability of the SM4 S-box is 2^-6, when the minimum active S-boxes reach 22 the differential characteristics will have probability 2^132, which is higher than enumeration (2^128).

This indicates that 19 rounds and 23 rounds under related key and single key settings will provide a minimum of 22 active S-boxes and is able to resist related-key differential attacks.

11.11. Summary of SM4 Cryptanalytic Attacks

Table 2 provides a summary on the strongest attacks on SM4 at the time of publishing.

Method	Rounds	Complexity	Reference
Linear	24	Time: 2^{122.6}, Data: 2^{122.6}, Memory: 2^{85}	[SM4-NLC]
Multi-dimensional Linear	23	Time: 2^{122.7}, Data: 2^{122.6}, Memory: 2^{120.6}	[SM4-LinearLiu]
Differential	23	Time: 2^{126.7}, Data: 2^{117}, Memory: 2^{120.7}	[SM4-DiffSu]
Matrix	18	Time: 2^{110.77}, Data: 2^{127}, Memory 2^{130}	[SM4-MatrixShe]
Impossible Differential	17	Time: 2^{132}, Data: 2^{117}, Memory: --	[SM4-IDCWang]
Zero-correlation Linear	14	Time: 2^{120.7}, Data: 2^{123.5}, Memory: 2^{73}	[SM4-ZCLC]
Integral	14	Time: 2^{96.5}, Data: 2^{32}, Memory: --	[SM4-ICZhong]

Table 2: Leading SM4 Attacks As Of Publication

As of the publication of this document, no open research results have provided a method to successfully attack beyond 24 rounds of SM4.

The traditional view suggests that SM4 provides an extra safety margin compared to blockciphers adopted in [ISO.IEC.18033-3] that already have full-round attacks, including MISTY1 [MISTY1-IC] [MISTY1-270] and AES [AES-CA] [AES-BC] [AES-RKC].

12. Security Considerations

- o Products and services that utilize cryptography are regulated by the SCA [SCA]; they must be explicitly approved or certified by the SCA before being allowed to be sold or used in China.

o SM4 is a blockcipher symmetric algorithm with key length of 128 bits. It is considered as an alternative to AES-128 [NIST.FIPS.197].

o SM4 [GBT.32907-2016] is a blockcipher certified by the SCA [SCA]. No formal proof of security is provided. There are no known practical attacks against SM4 algorithm by the time of publishing this document, but there are security concerns with regards to side-channel attacks when the SM4 algorithm is implemented in hardware.
  For instance, [SM4-Power] illustrated an attack by measuring the power consumption of the device. A chosen ciphertext attack, assuming a fixed correlation between the round keys and data mask, is able to recover the round key successfully.
  When the SM4 algorithm is implemented in hardware, the parameters and keys *SHOULD* be randomly generated without fixed correlation.
  There have also been improvements to the hardware embodiment design for SM4 [SM4-VLSI] [SM4-FPGA], white-box implementions [SM4-WhiteBox], and performance enhancements [SM4-HiSpeed], that may resist such attacks.

o The IV does not have to be secret. The IV itself, or criteria enough to determine it, *MAY* be transmitted with ciphertext.

o SM4-ECB: ECB is one of the four original modes defined for DES. With its problem well known to "leak quite a large amount of information" [BC-EVAL], it *SHOULD NOT* be used in most cases.

o SM4-CBC, SM4-CFB, SM4-OFB: CBC, CFB and OFB are IV-based modes of operation originally defined for DES.
  When using these modes of operation, the IV *SHOULD* be random to preserve message confidentiality [BC-EVAL]. It is shown in the same document that CBC, CFB, OFB, the variants #CBC, #CFB that utilize the recommendation of [NIST.SP.800-38A] to make CBC and CFB nonce-based, are SemCPA secure as probabilistic encryption schemes.
  Various attack scenarios have been described in [BC-EVAL] and these modes *SHOULD NOT* be used unless for compatibility reasons.

o SM4-CTR: CTR is considered to be the "best" mode of operation within [NIST.SP.800-38A] as it is considered SemCPA secure as a nonce-based encryption scheme, providing provable-security guarantees as good as the classic modes of operation (ECB, CBC, CFB, OFB) [BC-EVAL].
  Users with no need of authenticity, non-malleablility and chosen-ciphertext (CCA) security *MAY* utilize this mode of operation [BC-EVAL].

13. IANA Considerations

This document does not require any action by IANA.

14. References

14.1. Normative References

[GBT.32907-2016]
Standardization Administration of the People's Republic of China, "GB/T 32907-2016: Information security technology -- SM4 block cipher algorithm", August 2016, <http://www.gb688.cn/bzgk/gb/newGbInfo?hcno=7803DE42D3BC5E80B0C3E5D8E873D56A>.

[ISO.IEC.18033-3.AMD2]
International Organization for Standardization, "ISO/IEC WD1 18033-3/AMD2 -- Encryption algorithms -- Part 3: Block ciphers -- Amendment 2", June 2017, <https://www.iso.org/standard/54531.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

14.2. Informative References

[AES-BC] Bogdanov, A., Khovratovich, D., and C. Rechberger, "Biclique Cryptanalysis of the Full AES", 2011, <https://doi.org/10.1007/978-3-642-25385-0_19>.

[AES-CA] Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., and D. Whiting, "Improved Cryptanalysis of Rijndael", Jan 2002, <https://doi.org/10.1007/3-540-44706-7_15>.

[AES-RKC] Biryukov, A. and D. Khovratovich, "Related-Key Cryptanalysis of the Full AES-192 and AES-256", 2009, <https://doi.org/10.1007/978-3-642-10366-7_1>.

[AutoDC] Siwei, S., Hu, L., Wang, P., Qiao, K., Ma, X., and L. Song, "Automatic Security Evaluation and (Related-key) Differential Characteristic Search: Application to SIMON, PRESENT, LBlock, DES(L) and Other Bit-Oriented Block Ciphers", 2014, <https://doi.org/10.1007/978-3-662-45611-8_9>.

[BC-Design]
Wu, W., "Block Cipher Design and Analysis (in Chinese)", October 2009, <http://www.tup.tsinghua.edu.cn/booksCenter/book_03193701.html>.

[BC-EVAL] Rogaway, P., "Evaluation of Some Blockcipher Modes of Operation", February 2011, <http://web.cs.ucdavis.edu/rogaway/papers/modes.pdf>.

[BOTAN] Lloyd, J., "Botan: Crypto and TLS for C++11", October 2017, <https://botan.randombit.net>.

[GB.15629.11-2003]
Standardization Administration of the People's Republic of China, "Information technology -- Telecommunications and information exchange between systems -- Local and metropolitan area networks -- Specific requirements -- Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications", May 2003, <http://www.gb688.cn/bzgk/gb/newGbInfo?hcno=74B9DD11287E72408C19C4D3A360D1BD>.

[GMT-0002-2012]
Office of State Commercial Administration of China, "GM/T 0002-2012: SM4 block cipher algorithm", March 2012, <http://www.oscca.gov.cn/Column/Column_32.htm>.

[GMT-0006-2012]
Office of State Commercial Administration of China, "GM/T 0006-2012: Cryptographic Application Identifier Criterion Specification", March 2012, <http://www.oscca.gov.cn/Column/Column_32.htm>.

[ISO.IEC.18033-3]
International Organization for Standardization, "ISO/IEC 18033-3:2010 -- Encryption algorithms -- Part 3: Block ciphers", December 2017, <https://www.iso.org/standard/54531.html>.

[LSW-Bio] Sun, M., "Lv Shu Wang -- A life in cryptography", November 2010, <http://press.ustc.edu.cn/sites/default/files/fujian/field_fujian_multi/20120113/%E5%90%95%E8%BF%B0%E6%9C%9B%20%E5%AF%86%E7%A0%81%E4%B8%80%E6%A0%B7%E7%9A%84%E4%BA%BA%E7%94%9F.pdf>.

[MISTY1-270]
Bar-On, A. and N. Keller, "A 2^{70} Attack on the Full MISTY1", 2016, <https://doi.org/10.1007/978-3-662-53018-4_16>.

[MISTY1-IC]
Todo, Y., "Integral Cryptanalysis on Full MISTY1", 2015, <https://doi.org/10.1007/s00145-016-9240-x>.

[NIST.FIPS.197]
National Institute of Standards and Technology, "NIST FIPS 197: Advanced Encryption Standard (AES)", November 2001, <https://doi.org/10.6028/NIST.FIPS.197>.

[NIST.SP.800-38A]
Dworkin, M., "NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation -- Methods and Techniques", December 2001, <http://dx.doi.org/10.6028/NIST.SP.800-38A>.

[OPENSSL] OpenSSL Software Foundation, "OpenSSL: Cryptography and SSL/TLS Toolkit", October 2017, <https://www.openssl.org>.

[SCA] State Cryptography Administration of China, "State Cryptography Administration of China", Dec 2017, <http://www.sca.gov.cn>.

[SideChannel]
Lei, Q., Wu, L., Zhang, S., Zhang, X., Li, X., Pan, L., and Z. Dong, "Software Hardware Co-design for Side-Channel Analysis Platform on Security Chips", December 2015, <https://doi.org/10.1109/CIS.2015.102>.

[SM4] Office of State Commercial Administration of China, "SMS4 Cryptographic Algorithm For Wireless LAN Products", January 2006, <http://www.oscca.gov.cn/UpFile/200621016423197990.pdf>.

[SM4-AAEr]
Erickson, J., Ding, J., and C. Christensen, "Algebraic Cryptanalysis of SMS4: Groebner Basis Attack and SAT Attack Compared", 2010, <https://doi.org/10.1007/978-3-642-14423-3_6>.

[SM4-AAJi]
Wen, J., Lei, H., and H. Ou, "Algebraic Attack to SMS4 and the Comparison with AES", 2009, <https://doi.org/10.1109/IAS.2009.171>.

[SM4-Details]
Lu, S., Su, B., Peng, P., Miao, Y., and L. Huo, "Overview on SM4 Algorithm", October 2016, <http://ris.sic.gov.cn/EN/Y2016/V2/I11/995>.

[SM4-DiffSu]
Su, B., Wu, W., and W. Zhang, "Security of the SMS4 Block Cipher Against Differential Cryptanalysis", January 2011, <https://doi.org/10.1007/s11390-011-9420-y>.

[SM4-DiffZhang1]
Zhang, L., Zhang, W., and W. Wu, "Cryptanalysis of Reduced-Round SMS4 Block Cipher", July 2008, <https://doi.org/10.1007/978-3-540-70500-0_16>.

[SM4-DiffZhang2]
Zhang, W., Wu, W., Feng, D., and B. Su, "Some New Observations on the SMS4 Block Cipher in the Chinese WAPI Standard", 2009, <https://doi.org/10.1007/978-3-642-00843-6_28>.

[SM4-DiffZhang3]
Zhang, M., Liu, J., and X. Wang, "22-Round SMS4 Differential Cryptanalysis", 2010, <http://www.airitilibrary.com/Publication/alDetailedMesh?docid=05296579-201003-201004120040-201004120040-43-47>.

[SM4-En]   Diffie, W. and G. Ledin, "SMS4 Encryption Algorithm for Wireless Networks", May 2008, <https://eprint.iacr.org/2008/329>.

[SM4-FPGA]
Cheng, H., Zhai, S., Fang, L., Ding, Q., and C. Huang, "Improvements of SM4 Algorithm and Application in Ethernet Encryption System Based on FPGA", July 2014, <https://www.researchgate.net/publication/287081686_Improvements_of_SM4_algorithm_and_application_in_Ethernet_encryption_system_based_on_FPGA>.

[SM4-HiSpeed]
Lv, Q., Li, L., and Y. Cao, "High-speed Encryption Decryption System Based on SM4", July 2016, <http://dx.doi.org/10.14257/ijsia.2016.10.9.01>.

[SM4-ICLiu]
Liu, F., Ji, W., Hu, L., Ding, J., Lv, S., Pyshkin, A., and R. Weinmann, "Analysis of the SMS4 Block Cipher", 2007, <https://doi.org/10.1007/978-3-540-73458-1_13>.

[SM4-ICZhong]
Zhong, M., Hu, Y., and J. Chen, "14-Round Square Attack on Blockcipher SMS4", 2008, <http://www.cnki.com.cn/Article/CJFDTotal-XDKD200801019.htm>.

[SM4-IDCLu]
Lu, J., "Attacking Reduced-Round Versions of the SMS4 Block Cipher in the Chinese WAPI Standard", 2007, <https://doi.org/10.1007/978-3-540-77048-0_24>.

[SM4-IDCToz]
Toz, D. and O. Dunkelman, "Analysis of Two Attacks on Reduced-Round Versions of the SMS4", 2008, <https://doi.org/10.1007/978-3-540-88625-9_10>.

[SM4-IDCWang]
Wang, G., "Improved Impossible Differential Cryptanalysis on SMS4", October 2010, <https://doi.org/10.1109/CIS.2012.116>.

[SM4-LDA] Kim, T., Kim, J., Kim, S., and J. Sung, "Linear and Differential Cryptanalysis of Reduced SMS4 Block Cipher", June 2008, <https://eprint.iacr.org/2008/281>.

[SM4-LinearDong]
Dong, X., "Security Analysis of the blockciphers AES and SM4", 2011, <http://kns.cnki.net/KCMS/detail/detail.aspx?dbcode=CDFD&dbname=CDFD1214&filename=1013114416.nh>.

[SM4-LinearEtrog]
Etrog, J. and M. Robshaw, "The Cryptanalysis of Reduced-Round SMS4", 2009, <https://doi.org/10.1007/978-3-642-04159-4_4>.

[SM4-LinearLiu]
Liu, M. and J. Chen, "Improved Linear Attacks on the Chinese Block Cipher Standard", November 2014, <https://doi.org/10.1007/s11390-014-1495-9>.

[SM4-MatrixShe]
Ping, S., "Matrix Attack On Blockcipher SMS4", 2012, <http://cdmd.cnki.com.cn/Article/CDMD-10422-1012464969.htm>.

[SM4-MatrixWei]
Wei, H., Cui, H., and X. Lu, "Differential-Algebraic Analysis of the SMS4 Block Cipher", 2012, <http://www.cnki.com.cn/Article/CJFDTotal-CDDD201202017.htm>.

[SM4-MLCho]
Cho, J. and K. Nyberg, "Improved linear cryptanalysis of SM4 block cipher", 2010, <https://scholar.google.com.hk/scholar?cluster=13432379689578293076>.

[SM4-MLLiu]
Zhiqiang, L., Dawu, G., and Z. Jing, "Multiple Linear Cryptanalysis of Reduced-Round SMS4 Block Cipher", 2010, <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.215.8314>.

[SM4-NLC] Liu, Y., Liang, H., Wang, W., and M. Wang, "New Linear Cryptanalysis of Chinese Commercial Block Cipher Standard SM4", June 2017, <https://doi.org/10.1155/2017/1461520>.

[SM4-Power]
Du, Z., Wu, Z., Wang, M., and J. Rao, "Improved chosen-plaintext power analysis attack against SM4 at the round-output", October 2015, <http://dx.doi.org/10.6028/NIST.FIPS.180-4>.

[SM4-Random]
Zhang, L. and W. Wu, "Pseudorandomness and Super-pseudorandomness of a non-balanced Feistel Structure using compressed functions", January 2009, <http://www.cnki.com.cn/Article/CJFDTOTAL-JSJX200907008.htm>.

[SM4-RKDC]
Zhang, J., Wu, W., and Y. Zheng, "Security of SM4 Against (Related-Key) Differential Cryptanalysis", November 2016, <http://doi.org/10.1007/978-3-319-49151-6_5>.

[SM4-Sbox]
Liu, J., Wei, B., and X. Dai, "Cryptographic Properties of S-box in SMS4", January 2011, <http://www.cnki.com.cn/Article/CJFDTotal-JSJC200805057.htm>.

[SM4-SLC] Zhang, B. and J. Chenhui, "Practical security against linear cryptanalysis for SMS4-like ciphers with SP round function", 2012, <https://doi.org/10.1007/s11432-011-4448-8>.

[SM4-SLDC]
Zhang, M., Liu, Y., Liu, J., and X. Min, "Practically Secure against Differential Cryptanalysis for Block Cipher SMS4", 2011, <http://www.ajetr.org/vol15/no1/n09.pdf>.

[SM4-VLSI]
Yu, S., Li, K., Li, K., Qin, Y., and Z. Tong, "A VLSI implementation of an SM4 algorithm resistant to power analysis", July 2016, <https://doi.org/10.3233/JIFS-169011>.

[SM4-WhiteBox]
Bai, K. and C. Wu, "A secure white-box SM4 implementation", May 2008, <http://dx.doi.org/10.1002/sec.1394>.

[SM4-ZCLC]
Ma, M., Zhao, Y., Liu, Q., and F. Liu, "Multidimensional Zero-correlation Linear Cryptanalysis on SMS4 Algorithm", September 2015, <http://www.jcr.cacrnet.org.cn:8080/mmxb/CN/abstract/abstract105.shtml>.

Appendix A. Appendix A: Example Calculations

A.1. Examples From GB/T 32907-2016

A.1.1. Example 1 (GB/T 32907-2016 Example 1 Encryption)

This is example 1 provided by [GBT.32907-2016] to demonstrate encryption of a plaintext.

Plaintext:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

Encryption key:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

Status of the round key (rk_i) and round output (X_i) per round:

```
   rk_0  = F12186F9   X_4  = 27FAD345
   rk_1  = 41662B61   X_5  = A18B4CB2
   rk_2  = 5A6AB19A   X_6  = 11C1E22A
   rk_3  = 7BA92077   X_7  = CC13E2EE
   rk_4  = 367360F4   X_8  = F87C5BD5
   rk_5  = 776A0C61   X_9  = 33220757
   rk_6  = B6BB89B3   X_10 = 77F4C297
   rk_7  = 24763151   X_11 = 7A96F2EB
   rk_8  = A520307C   X_12 = 27DAC07F
   rk_9  = B7584DBD   X_13 = 42DD0F19
   rk_10 = C30753ED   X_14 = B8A5DA02
   rk_11 = 7EE55B57   X_15 = 907127FA
   rk_12 = 6988608C   X_16 = 8B952B83
   rk_13 = 30D895B7   X_17 = D42B7C59
   rk_14 = 44BA14AF   X_18 = 2FFC5831
   rk_15 = 104495A1   X_19 = F69E6888
   rk_16 = D120B428   X_20 = AF2432C4
   rk_17 = 73B55FA3   X_21 = ED1EC85E
   rk_18 = CC874966   X_22 = 55A3BA22
   rk_19 = 92244439   X_23 = 124B18AA
   rk_20 = E89E641F   X_24 = 6AE7725F
   rk_21 = 98CA015A   X_25 = F4CBA1F9
   rk_22 = C7159060   X_26 = 1DCDFA10
   rk_23 = 99E1FD2E   X_27 = 2FF60603
   rk_24 = B79BD80C   X_28 = EFF24FDC
   rk_25 = 1D2115B0   X_29 = 6FE46B75
   rk_26 = 0E228AEB   X_30 = 893450AD
   rk_27 = F1780C81   X_31 = 7B938F4C
   rk_28 = 428D3654   X_32 = 536E4246
   rk_29 = 62293496   X_33 = 86B3E94F
   rk_30 = 01CF72E5   X_34 = D206965E
   rk_31 = 9124A012   X_35 = 681EDF34
```

Ciphertext:

```
   68 1E DF 34 D2 06 96 5E 86 B3 E9 4F 53 6E 42 46
```

A.1.2. Example 2 (GB/T 32907-2016 Example 1 Decryption)

This demonstrates the decryption process of the Example 1 ciphertext provided by [GBT.32907-2016].

Ciphertext:

```
   68 1E DF 34 D2 06 96 5E 86 B3 E9 4F 53 6E 42 46
```

Encryption key:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

Status of the round key (rk_i) and round output (X_i) per round:

```
rk_31 = 9124A012    X_35 = 7B938F4C
rk_30 = 01CF72E5    X_34 = 893450AD
rk_29 = 62293496    X_33 = 6FE46B75
rk_28 = 428D3654    X_32 = EFF24FDC
rk_27 = F1780C81    X_31 = 2FF60603
rk_26 = 0E228AEB    X_30 = 1DCDFA10
rk_25 = 1D2115B0    X_29 = F4CBA1F9
rk_24 = B79BD80C    X_28 = 6AE7725F
rk_23 = 99E1FD2E    X_27 = 124B18AA
rk_22 = C7159060    X_26 = 55A3BA22
rk_21 = 98CA015A    X_25 = ED1EC85E
rk_20 = E89E641F    X_24 = AF2432C4
rk_19 = 92244439    X_23 = F69E6888
rk_18 = CC874966    X_22 = 2FFC5831
rk_17 = 73B55FA3    X_21 = D42B7C59
rk_16 = D120B428    X_20 = 8B952B83
rk_15 = 104495A1    X_19 = 907127FA
rk_14 = 44BA14AF    X_18 = B8A5DA02
rk_13 = 30D895B7    X_17 = 42DD0F19
rk_12 = 6988608C    X_16 = 27DAC07F
rk_11 = 7EE55B57    X_15 = 7A96F2EB
rk_10 = C30753ED    X_14 = 77F4C297
rk_9  = B7584DBD    X_13 = 33220757
rk_8  = A520307C    X_12 = F87C5BD5
rk_7  = 24763151    X_11 = CC13E2EE
rk_6  = B6BB89B3    X_10 = 11C1E22A
rk_5  = 776A0C61    X_9  = A18B4CB2
rk_4  = 367360F4    X_8  = 27FAD345
rk_3  = 7BA92077    X_7  = 76543210
rk_2  = 5A6AB19A    X_6  = FEDCBA98
rk_1  = 41662B61    X_5  = 89ABCDEF
rk_0  = F12186F9    X_4  = 01234567
```

Plaintext:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

A.1.3. Example 3 (GB/T 32907-2016 Example 2 Encryption)

This example is provided by [GBT.32907-2016] to demonstrate encryption of a plaintext 1,000,000 times repeatedly, using a fixed encryption key.

Plaintext:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

Encryption Key:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

Ciphertext:

```
59 52 98 C7 C6 FD 27 1F 04 02 F8 04 C3 3D 3F 66
```

A.1.4. Example 4

The following example demonstrates encryption of a different message using a different key from the above examples.

Plaintext:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Encryption key:

```
FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF
```

Status of the round key (rk_i) and round output (X_i) per round:

```
   rk_0  = 0D8CC1B4    X_4  = F7EAEB6A
   rk_1  = AC44F213    X_5  = B4967C0F
   rk_2  = 188C0C40    X_6  = 5B9B2419
   rk_3  = 7537585E    X_7  = F46BECBA
   rk_4  = 627646F5    X_8  = A8013E25
   rk_5  = 54D785AD    X_9  = B38E2ABE
   rk_6  = 51B96DEE    X_10 = 3E7C99A1
   rk_7  = 0C385958    X_11 = 6DD5F47F
   rk_8  = 5E494992    X_12 = B286430C
   rk_9  = 32F3FE04    X_13 = AB997DE3
   rk_10 = 3A3A733D    X_14 = 80F8F21F
   rk_11 = 0EDFB91D    X_15 = 4EF7052E
   rk_12 = 6823CD6B    X_16 = 4462FFAF
   rk_13 = 40F7D825    X_17 = 14DFD5EA
   rk_14 = 4BD68EE5    X_18 = 6D33EFED
   rk_15 = 165A36C8    X_19 = 3A4F8B3C
   rk_16 = 56608984    X_20 = 1A435088
   rk_17 = 23F35FF4    X_21 = 4E64B153
   rk_18 = 8B592B3E    X_22 = 0415CEDA
   rk_19 = 80F7388A    X_23 = ADD88955
   rk_20 = 0415C409    X_24 = 73964EF1
   rk_21 = AFDF1370    X_25 = B0085092
   rk_22 = CF444772    X_26 = 554A1293
   rk_23 = 9AF9901F    X_27 = 4BC6D6A8
   rk_24 = C457578C    X_28 = 7BB650E1
   rk_25 = 95701C60    X_29 = DDFB8A61
   rk_26 = 2B0F4EE1    X_30 = 5C4DFD78
   rk_27 = 7F826139    X_31 = FD9066FD
   rk_28 = FA37F8D9    X_32 = 55ADB594
   rk_29 = D18AF8CE    X_33 = AC1B3EA9
   rk_30 = 5BD5D8C6    X_34 = 13F01ADE
   rk_31 = 711138B7    X_35 = F766678F
```

Ciphertext:

```
   F7 66 67 8F 13 F0 1A DE AC 1B 3E A9 55 AD B5 94
```

A.1.5. Example 5

The following example demonstrates decryption of Example 4.

Ciphertext:

```
   F7 66 67 8F 13 F0 1A DE AC 1B 3E A9 55 AD B5 94
```

Encryption key:

```
   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF
```

Status of the round key (rk_i) and round output (X_i) per round:

```
rk_31 = 711138B7    X_35 = FD9066FD
rk_30 = 5BD5D8C6    X_34 = 5C4DFD78
rk_29 = D18AF8CE    X_33 = DDFB8A61
rk_28 = FA37F8D9    X_32 = 7BB650E1
rk_27 = 7F826139    X_31 = 4BC6D6A8
rk_26 = 2B0F4EE1    X_30 = 554A1293
rk_25 = 95701C60    X_29 = B0085092
rk_24 = C457578C    X_28 = 73964EF1
rk_23 = 9AF9901F    X_27 = ADD88955
rk_22 = CF444772    X_26 = 0415CEDA
rk_21 = AFDF1370    X_25 = 4E64B153
rk_20 = 0415C409    X_24 = 1A435088
rk_19 = 80F7388A    X_23 = 3A4F8B3C
rk_18 = 8B592B3E    X_22 = 6D33EFED
rk_17 = 23F35FF4    X_21 = 14DFD5EA
rk_16 = 56608984    X_20 = 4462FFAF
rk_15 = 165A36C8    X_19 = 4EF7052E
rk_14 = 4BD68EE5    X_18 = 80F8F21F
rk_13 = 40F7D825    X_17 = AB997DE3
rk_12 = 6823CD6B    X_16 = B286430C
rk_11 = 0EDFB91D    X_15 = 6DD5F47F
rk_10 = 3A3A733D    X_14 = 3E7C99A1
rk_9  = 32F3FE04    X_13 = B38E2ABE
rk_8  = 5E494992    X_12 = A8013E25
rk_7  = 0C385958    X_11 = F46BECBA
rk_6  = 51B96DEE    X_10 = 5B9B2419
rk_5  = 54D785AD    X_9  = B4967C0F
rk_4  = 627646F5    X_8  = F7EAEB6A
rk_3  = 7537585E    X_7  = 0C0D0E0F
rk_2  = 188C0C40    X_6  = 08090A0B
rk_1  = AC44F213    X_5  = 04050607
rk_0  = 0D8CC1B4    X_4  = 00010203
```

Plaintext:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

A.1.6. Example 6

This example is based on Example 4 to demonstrate encryption of a plaintext 1,000,000 times repeatedly, using a fixed encryption key.

Plaintext:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Encryption Key:

```
FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF
```

Ciphertext:

```
37 9A 96 D0 A6 A5 A5 06 0F B4 60 C7 5D 18 79 ED
```

A.2. Examples For Various Modes Of Operations

The following examples can be verified using open-source cryptographic libraries including:

- o the Botan cryptographic library [BOTAN] with SM4 support, and
- o the OpenSSL Cryptography and SSL/TLS Toolkit [OPENSSL] with SM4 support

A.2.1. SM4-ECB Examples

A.2.1.1. Example 1

Plaintext:

```
AA AA AA AA BB BB BB BB CC CC CC CC DD DD DD DD
EE EE EE EE FF FF FF FF AA AA AA AA BB BB BB BB
```

Encryption Key:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

Ciphertext:

```
5E C8 14 3D E5 09 CF F7 B5 17 9F 8F 47 4B 86 19
2F 1D 30 5A 7F B1 7D F9 85 F8 1C 84 82 19 23 04
```

A.2.1.2. Example 2

Plaintext:

```
AA AA AA AA BB BB BB BB CC CC CC CC DD DD DD DD
EE EE EE EE FF FF FF FF AA AA AA AA BB BB BB BB
```

Encryption Key:

```
FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF
```

Ciphertext:

```
   C5 87 68 97 E4 A5 9B BB A7 2A 10 C8 38 72 24 5B
   12 DD 90 BC 2D 20 06 92 B5 29 A4 15 5A C9 E6 00
```

A.2.2. SM4-CBC Examples

A.2.2.1. Example 1

```
   Plaintext:

   AA AA AA AA BB BB BB BB CC CC CC CC DD DD DD DD
   EE EE EE EE FF FF FF FF AA AA AA AA BB BB BB BB

   Encryption Key:

   01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10

   IV:

   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

   Ciphertext:

   78 EB B1 1C C4 0B 0A 48 31 2A AE B2 04 02 44 CB
   4C B7 01 69 51 90 92 26 97 9B 0D 15 DC 6A 8F 6D
```

A.2.2.2. Example 2

```
   Plaintext:

   AA AA AA AA BB BB BB BB CC CC CC CC DD DD DD DD
   EE EE EE EE FF FF FF FF AA AA AA AA BB BB BB BB

   Encryption Key:

   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

   IV:

   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

   Ciphertext:

   0D 3A 6D DC 2D 21 C6 98 85 72 15 58 7B 7B B5 9A
   91 F2 C1 47 91 1A 41 44 66 5E 1F A1 D4 0B AE 38
```

A.2.3. SM4-OFB Examples

A.2.3.1. Example 1

Plaintext:

```
AA AA AA AA BB BB BB BB CC CC CC CC DD DD DD DD
EE EE EE EE FF FF FF FF AA AA AA AA BB BB BB BB
```

Encryption Key:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

IV:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Ciphertext:

```
AC 32 36 CB 86 1D D3 16 E6 41 3B 4E 3C 75 24 B7
1D 01 AC A2 48 7C A5 82 CB F5 46 3E 66 98 53 9B
```

A.2.3.2. Example 2

Plaintext:

```
AA AA AA AA BB BB BB BB CC CC CC CC DD DD DD DD
EE EE EE EE FF FF FF FF AA AA AA AA BB BB BB BB
```

Encryption Key:

```
FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF
```

IV:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Ciphertext:

```
5D CC CD 25 A8 4B A1 65 60 D7 F2 65 88 70 68 49
33 FA 16 BD 5C D9 C8 56 CA CA A1 E1 01 89 7A 97
```

A.2.4. SM4-CFB Examples

A.2.4.1. Example 1

Plaintext:

```
AA AA AA AA BB BB BB BB CC CC CC CC DD DD DD DD
EE EE EE EE FF FF FF FF AA AA AA AA BB BB BB BB
```

Encryption Key:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

IV:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Ciphertext:

```
AC 32 36 CB 86 1D D3 16 E6 41 3B 4E 3C 75 24 B7
69 D4 C5 4E D4 33 B9 A0 34 60 09 BE B3 7B 2B 3F
```

A.2.4.2. Example 2

Plaintext:

```
AA AA AA AA BB BB BB BB CC CC CC CC DD DD DD DD
EE EE EE EE FF FF FF FF AA AA AA AA BB BB BB BB
```

Encryption Key:

```
FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF
```

IV:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Ciphertext:

```
5D CC CD 25 A8 4B A1 65 60 D7 F2 65 88 70 68 49
0D 9B 86 FF 20 C3 BF E1 15 FF A0 2C A6 19 2C C5
```

A.2.5. SM4-CTR Examples

A.2.5.1. Example 1

Plaintext:

```
AA AA AA AA AA AA AA AA BB BB BB BB BB BB BB BB
CC CC CC CC CC CC CC CC DD DD DD DD DD DD DD DD
EE EE EE EE EE EE EE EE FF FF FF FF FF FF FF FF
AA AA AA AA AA AA AA AA BB BB BB BB BB BB BB BB
```

Encryption Key:

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

IV:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Ciphertext:

```
AC 32 36 CB 97 0C C2 07 91 36 4C 39 5A 13 42 D1
A3 CB C1 87 8C 6F 30 CD 07 4C CE 38 5C DD 70 C7
F2 34 BC 0E 24 C1 19 80 FD 12 86 31 0C E3 7B 92
6E 02 FC D0 FA A0 BA F3 8B 29 33 85 1D 82 45 14
```

A.2.5.2. Example 2

Plaintext:

```
AA AA AA AA AA AA AA AA BB BB BB BB BB BB BB BB
CC CC CC CC CC CC CC CC DD DD DD DD DD DD DD DD
EE EE EE EE EE EE EE EE FF FF FF FF FF FF FF FF
AA AA AA AA AA AA AA AA BB BB BB BB BB BB BB BB
```

Encryption Key:

```
FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF
```

IV:

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
```

Ciphertext:

```
5D CC CD 25 B9 5A B0 74 17 A0 85 12 EE 16 0E 2F
8F 66 15 21 CB BA B4 4C C8 71 38 44 5B C2 9E 5C
0A E0 29 72 05 D6 27 04 17 3B 21 23 9B 88 7F 6C
8C B5 B8 00 91 7A 24 88 28 4B DE 9E 16 EA 29 06
```

Appendix B. Sample Implementation In C

B.1. sm4.h

"sm4.h" is the header file for the SM4 function.

```
<CODE BEGINS>
#ifndef HEADER_SM4_H
# define HEADER_SM4_H

#include <inttypes.h>

# define SM4_BLOCK_SIZE    16
# define SM4_KEY_SCHEDULE  32

void sm4_encrypt(uint8_t key[],
    unsigned char plaintext[],
    unsigned char ciphertext[]);

void sm4_decrypt(uint8_t key[],
    unsigned char ciphertext[],
    unsigned char plaintext[]);

#endif

<CODE ENDS>
```

B.2. sm4.c

"sm4.c" contains the main implementation of SM4.

```
<CODE BEGINS>
/* A sample implementation of SM4 */

#include <stdlib.h>
#include <string.h>
#include "sm4.h"
#include "print.h"

/* Operations */
/* Rotate Left 32-bit number */
#define ROTL32(X, n) (((X) << (n)) | ((X) >> (32 - (n))))

static uint32_t sm4_ck[32] = {
  0x00070E15, 0x1C232A31, 0x383F464D, 0x545B6269,
  0x70777E85, 0x8C939AA1, 0xA8AFB6BD, 0xC4CBD2D9,
  0xE0E7EEF5, 0xFC030A11, 0x181F262D, 0x343B4249,
  0x50575E65, 0x6C737A81, 0x888F969D, 0xA4ABB2B9,
```

```
     0xC0C7CED5, 0xDCE3EAF1, 0xF8FF060D, 0x141B2229,
     0x30373E45, 0x4C535A61, 0x686F767D, 0x848B9299,
     0xA0A7AEB5, 0xBCC3CAD1, 0xD8DFE6ED, 0xF4FB0209,
     0x10171E25, 0x2C333A41, 0x484F565D, 0x646B7279
   };

   static uint8_t sm4_sbox[256] = {
     0xD6, 0x90, 0xE9, 0xFE, 0xCC, 0xE1, 0x3D, 0xB7,
     0x16, 0xB6, 0x14, 0xC2, 0x28, 0xFB, 0x2C, 0x05,
     0x2B, 0x67, 0x9A, 0x76, 0x2A, 0xBE, 0x04, 0xC3,
     0xAA, 0x44, 0x13, 0x26, 0x49, 0x86, 0x06, 0x99,
     0x9C, 0x42, 0x50, 0xF4, 0x91, 0xEF, 0x98, 0x7A,
     0x33, 0x54, 0x0B, 0x43, 0xED, 0xCF, 0xAC, 0x62,
     0xE4, 0xB3, 0x1C, 0xA9, 0xC9, 0x08, 0xE8, 0x95,
     0x80, 0xDF, 0x94, 0xFA, 0x75, 0x8F, 0x3F, 0xA6,
     0x47, 0x07, 0xA7, 0xFC, 0xF3, 0x73, 0x17, 0xBA,
     0x83, 0x59, 0x3C, 0x19, 0xE6, 0x85, 0x4F, 0xA8,
     0x68, 0x6B, 0x81, 0xB2, 0x71, 0x64, 0xDA, 0x8B,
     0xF8, 0xEB, 0x0F, 0x4B, 0x70, 0x56, 0x9D, 0x35,
     0x1E, 0x24, 0x0E, 0x5E, 0x63, 0x58, 0xD1, 0xA2,
     0x25, 0x22, 0x7C, 0x3B, 0x01, 0x21, 0x78, 0x87,
     0xD4, 0x00, 0x46, 0x57, 0x9F, 0xD3, 0x27, 0x52,
     0x4C, 0x36, 0x02, 0xE7, 0xA0, 0xC4, 0xC8, 0x9E,
     0xEA, 0xBF, 0x8A, 0xD2, 0x40, 0xC7, 0x38, 0xB5,
     0xA3, 0xF7, 0xF2, 0xCE, 0xF9, 0x61, 0x15, 0xA1,
     0xE0, 0xAE, 0x5D, 0xA4, 0x9B, 0x34, 0x1A, 0x55,
     0xAD, 0x93, 0x32, 0x30, 0xF5, 0x8C, 0xB1, 0xE3,
     0x1D, 0xF6, 0xE2, 0x2E, 0x82, 0x66, 0xCA, 0x60,
     0xC0, 0x29, 0x23, 0xAB, 0x0D, 0x53, 0x4E, 0x6F,
     0xD5, 0xDB, 0x37, 0x45, 0xDE, 0xFD, 0x8E, 0x2F,
     0x03, 0xFF, 0x6A, 0x72, 0x6D, 0x6C, 0x5B, 0x51,
     0x8D, 0x1B, 0xAF, 0x92, 0xBB, 0xDD, 0xBC, 0x7F,
     0x11, 0xD9, 0x5C, 0x41, 0x1F, 0x10, 0x5A, 0xD8,
     0x0A, 0xC1, 0x31, 0x88, 0xA5, 0xCD, 0x7B, 0xBD,
     0x2D, 0x74, 0xD0, 0x12, 0xB8, 0xE5, 0xB4, 0xB0,
     0x89, 0x69, 0x97, 0x4A, 0x0C, 0x96, 0x77, 0x7E,
     0x65, 0xB9, 0xF1, 0x09, 0xC5, 0x6E, 0xC6, 0x84,
     0x18, 0xF0, 0x7D, 0xEC, 0x3A, 0xDC, 0x4D, 0x20,
     0x79, 0xEE, 0x5F, 0x3E, 0xD7, 0xCB, 0x39, 0x48
   };

   static uint32_t sm4_fk[4] = {
     0xA3B1BAC6, 0x56AA3350, 0x677D9197, 0xB27022DC
   };

   static uint32_t load_u32_be(const uint8_t *b, uint32_t n)
   {
     return ((uint32_t)b[4 * n + 3] << 24) |
```

```
            ((uint32_t)b[4 * n + 2] << 16) |
            ((uint32_t)b[4 * n + 1] << 8)  |
            ((uint32_t)b[4 * n    ]);
   }

   static void store_u32_be(uint32_t v, uint8_t *b)
   {
     b[3] = (uint8_t)(v >> 24);
     b[2] = (uint8_t)(v >> 16);
     b[1] = (uint8_t)(v >> 8);
     b[0] = (uint8_t)(v);
   }

   static void sm4_key_schedule(uint8_t key[], uint32_t rk[])
   {
     uint32_t t, x, k[36];
     int i;

     for (i = 0; i < 4; i++)
     {
       k[i] = load_u32_be(key, i) ^ sm4_fk[i];
     }

     /* T' */
     for (i = 0; i < SM4_KEY_SCHEDULE; ++i)
     {
       x = k[i + 1] ^ k[i + 2] ^ k[i + 3] ^ sm4_ck[i];

       /* Nonlinear operation tau */
       t = ((uint32_t)sm4_sbox[(uint8_t)(x >> 24)]) << 24 |
           ((uint32_t)sm4_sbox[(uint8_t)(x >> 16)]) << 16 |
           ((uint32_t)sm4_sbox[(uint8_t)(x >>  8)]) <<  8 |
           ((uint32_t)sm4_sbox[(uint8_t)(x)]);

       /* Linear operation L' */
       k[i+4] = k[i] ^ (t ^ ROTL32(t, 13) ^ ROTL32(t, 23));
       rk[i] = k[i + 4];
     }


   }

   #define SM4_ROUNDS(k0, k1, k2, k3, F)   \
     do {                                 \
       X0 ^= F(X1 ^ X2 ^ X3 ^ rk[k0]); \
       X1 ^= F(X0 ^ X2 ^ X3 ^ rk[k1]); \
       X2 ^= F(X0 ^ X1 ^ X3 ^ rk[k2]); \
       X3 ^= F(X0 ^ X1 ^ X2 ^ rk[k3]); \
```

```
        debug_print("rk_%0.2i = %0.8x  " \
          "  X_%0.2i = %0.8x\n", k0, rk[k0], k0+4, X0); \
        debug_print("rk_%0.2i = %0.8x  " \
          "  X_%0.2i = %0.8x\n", k1, rk[k1], k1+4, X1); \
        debug_print("rk_%0.2i = %0.8x  " \
          "  X_%0.2i = %0.8x\n", k2, rk[k2], k2+4, X2); \
        debug_print("rk_%0.2i = %0.8x  " \
          "  X_%0.2i = %0.8x\n", k3, rk[k3], k3+4, X3); \
      } while(0)

   static uint32_t sm4_t(uint32_t x)
   {
     uint32_t t = 0;

     t |= ((uint32_t)sm4_sbox[(uint8_t)(x >> 24)]) << 24;
     t |= ((uint32_t)sm4_sbox[(uint8_t)(x >> 16)]) << 16;
     t |= ((uint32_t)sm4_sbox[(uint8_t)(x >> 8)]) << 8;
     t |= sm4_sbox[(uint8_t)x];

     /*
      * L linear transform
      */
     return t ^ ROTL32(t, 2) ^ ROTL32(t, 10) ^
            ROTL32(t, 18) ^ ROTL32(t, 24);
   }

   void sm4_encrypt(uint8_t key[],
       unsigned char plaintext[],
       unsigned char ciphertext[])
   {
     uint32_t rk[SM4_KEY_SCHEDULE], X0, X1, X2, X3;
     int i, j;

     sm4_key_schedule(key, rk);

     X0 = load_u32_be(plaintext, 0);
     X1 = load_u32_be(plaintext, 1);
     X2 = load_u32_be(plaintext, 2);
     X3 = load_u32_be(plaintext, 3);

     SM4_ROUNDS( 0,  1,  2,  3, sm4_t);
     SM4_ROUNDS( 4,  5,  6,  7, sm4_t);
     SM4_ROUNDS( 8,  9, 10, 11, sm4_t);
     SM4_ROUNDS(12, 13, 14, 15, sm4_t);
     SM4_ROUNDS(16, 17, 18, 19, sm4_t);
     SM4_ROUNDS(20, 21, 22, 23, sm4_t);
     SM4_ROUNDS(24, 25, 26, 27, sm4_t);
     SM4_ROUNDS(28, 29, 30, 31, sm4_t);
```

```
     store_u32_be(X3, ciphertext);
     store_u32_be(X2, ciphertext + 4);
     store_u32_be(X1, ciphertext + 8);
     store_u32_be(X0, ciphertext + 12);
   }

   void sm4_decrypt(uint8_t key[],
       unsigned char ciphertext[],
       unsigned char plaintext[])
   {
     uint32_t rk[SM4_KEY_SCHEDULE], X0, X1, X2, X3;
     int i, j;

     sm4_key_schedule(key, rk);

     X0 = load_u32_be(ciphertext, 0);
     X1 = load_u32_be(ciphertext, 1);
     X2 = load_u32_be(ciphertext, 2);
     X3 = load_u32_be(ciphertext, 3);

     SM4_ROUNDS(31, 30, 29, 28, sm4_t);
     SM4_ROUNDS(27, 26, 25, 24, sm4_t);
     SM4_ROUNDS(23, 22, 21, 20, sm4_t);
     SM4_ROUNDS(19, 18, 17, 16, sm4_t);
     SM4_ROUNDS(15, 14, 13, 12, sm4_t);
     SM4_ROUNDS(11, 10,  9,  8, sm4_t);
     SM4_ROUNDS( 7,  6,  5,  4, sm4_t);
     SM4_ROUNDS( 3,  2,  1,  0, sm4_t);

     store_u32_be(X3, plaintext);
     store_u32_be(X2, plaintext + 4);
     store_u32_be(X1, plaintext + 8);
     store_u32_be(X0, plaintext + 12);
   }

   <CODE ENDS>
```

B.3. sm4_main.c

"sm4_main.c" is used to run the examples provided in this document and print out internal state for implementation reference.

```
   <CODE BEGINS>
   #include <stdlib.h>
   #include <string.h>
   #include <stdbool.h>
   #include "sm4.h"
   #include "print.h"
```

```
   typedef struct {
     unsigned char* key;
     unsigned char* message;
     unsigned char* expected;
     int iterations;
     bool encrypt;
   } test_case;

   int sm4_run_example(test_case tc)
   {
     unsigned char input[SM4_BLOCK_SIZE] = {0};
     unsigned char output[SM4_BLOCK_SIZE] = {0};
     int i;

     debug_print("-----------------------"
         " Message Input m Begin "
         "-------------------------\n");
     print_bytes((unsigned int*)tc.message, SM4_BLOCK_SIZE);
     debug_print("----------------------- "
         "Message Input m End "
         "---------------------------\n");

     if (tc.encrypt)
     {
       debug_print("----------------------- "
           "Encrypt "
           "---------------------------\n");
       memcpy(input, tc.message, SM4_BLOCK_SIZE);
       for (i = 0; i != tc.iterations; ++i)
       {
         sm4_encrypt(tc.key,
             (unsigned char*)input,
             (unsigned char*)output);
         memcpy(input, output, SM4_BLOCK_SIZE);
       }
     }
     else
     {
       debug_print("----------------------- "
           "Decrypt "
           "---------------------------\n");
       memcpy(input, tc.message, SM4_BLOCK_SIZE);
       for (i = 0; i != tc.iterations; ++i)
       {
         sm4_decrypt(tc.key,
             (unsigned char*)input,
             (unsigned char*)output);
         memcpy(input, output, SM4_BLOCK_SIZE);
```

```
       }
     }

     debug = 1;
     debug_print("+++++++++++++++++++++++++++++++"
         " RESULT "
         "++++++++++++++++++++++++++++++++\n");
     debug_print("RESULTS:\n");
     debug_print(" Expected:\n");
     print_bytes((unsigned int*)tc.expected, SM4_BLOCK_SIZE);

     debug_print(" Output:\n");
     print_bytes((unsigned int*)output, SM4_BLOCK_SIZE);

     debug = 0;
     return memcmp(
       (unsigned char*)output,
       (unsigned char*)tc.expected,
       SM4_BLOCK_SIZE
     );
   }

   int main(int argc, char **argv)
   {

     int i;
     unsigned char key[SM4_BLOCK_SIZE];
     unsigned char block[SM4_BLOCK_SIZE];

     test_case tests[8] = {0};

     /*
      * This test vector comes from Example 1 of GB/T 32907-2016,
      */
     static const unsigned int gbt32907k1[SM4_BLOCK_SIZE] = {
       0x01234567, 0x89abcdef,
       0xfedcba98, 0x76543210
     };
     static const unsigned int gbt32907m1[SM4_BLOCK_SIZE] = {
       0x01234567, 0x89abcdef,
       0xfedcba98, 0x76543210
     };
     static const unsigned int gbt32907e1[SM4_BLOCK_SIZE] = {
       0x681edf34, 0xd206965e,
       0x86b3e94f, 0x536e4246
     };
     test_case gbt32907t1 = {
       (unsigned char*)gbt32907k1,
```

```
       (unsigned char*)gbt32907m1,
       (unsigned char*)gbt32907e1,
       1,
       true
     };
     tests[0] = gbt32907t1;

     /*
      * This test vector comes from Example 2 from GB/T 32907-2016.
      * After 1,000,000 iterations.
      */
     static const unsigned int gbt32907e2[SM4_BLOCK_SIZE] = {
       0x595298c7, 0xc6fd271f,
       0x0402f804, 0xc33d3f66
     };
     test_case gbt32907t2 = {
       (unsigned char*)gbt32907k1,
       (unsigned char*)gbt32907m1,
       (unsigned char*)gbt32907e2,
       1000000,
       true
     };
     tests[1] = gbt32907t2;

     /*
      * This test vector reverses Example 1 of GB/T 32907-2016.
      * After decrypting 1 iteration.
      */
     test_case gbt32907t3 = {
       (unsigned char*)gbt32907k1,
       (unsigned char*)gbt32907e1,
       (unsigned char*)gbt32907m1,
       1,
       false
     };
     tests[2] = gbt32907t3;

     /*
      * This test vector reverses Example 2 of GB/T 32907-2016.
      * After decrypting 1,000,000 iterations.
      */
     test_case gbt32907t4 = {
       (unsigned char*)gbt32907k1,
       (unsigned char*)gbt32907e2,
       (unsigned char*)gbt32907m1,
       1000000,
       false
     };
```

```
     tests[3] = gbt32907t4;

     /*
      * Newly added examples to demonstrate key changes.
      */
     static const unsigned int newexamplek1[SM4_BLOCK_SIZE] = {
       0xfedcba98, 0x76543210,
       0x01234567, 0x89abcdef
     };
     static const unsigned int newexamplem1[SM4_BLOCK_SIZE] = {
       0x00010203, 0x04050607,
       0x08090a0b, 0x0c0d0e0f
     };
     static const unsigned int newexamplee1[SM4_BLOCK_SIZE] = {
       0xf766678f, 0x13f01ade,
       0xac1b3ea9, 0x55adb594
     };
     /*
      */
     test_case newexample1 = {
       (unsigned char*)newexamplek1,
       (unsigned char*)newexamplem1,
       (unsigned char*)newexamplee1,
       1,
       true
     };
     tests[4] = newexample1;

     test_case newexample2 = {
       (unsigned char*)newexamplek1,
       (unsigned char*)newexamplee1,
       (unsigned char*)newexamplem1,
       1,
       false
     };
     tests[5] = newexample2;

     /*
      * After 1,000,000 iterations.
      */
     static const unsigned int newexamplee2[SM4_BLOCK_SIZE] = {
       0x379a96d0, 0xa6a5a506,
       0x0fb460c7, 0x5d1879ed
     };
     test_case newexample3 = {
       (unsigned char*)newexamplek1,
       (unsigned char*)newexamplem1,
       (unsigned char*)newexamplee2,
```

```
        1000000,
        true
      };
      tests[6] = newexample3;


      for (i = 0; i < 7; ++i)
      {

        if (i == 1 || i == 3)
          continue;

        printf("sm4_example[%2i]: %s\n", i,
          sm4_run_example(tests[i]) ?  "FAIL" : "PASS");
      }

      return 0;
    }

    <CODE ENDS>
```

B.4. print.c and print.h

"print.c" and "print.h" are used to provide pretty formatting used to print out the examples for this document.

"print.h"

```
<CODE BEGINS>
#ifndef SM4PRINT_H
#define SM4PRINT_H

#define DEBUG 0
#define debug_print(...) \
  do { if (DEBUG) fprintf(stderr, __VA_ARGS__); } while (0)

#include <stdio.h>

void print_bytes(unsigned* buf, int n);

#endif
<CODE ENDS>
```

"print.c"

```
   <CODE BEGINS>
   #include <stdio.h>
   #include "print.h"

   void print_bytes(unsigned int* buf, int n)
   {
     unsigned char* ptr = (unsigned char*)buf;
     int i, j;

     for (i = 0; i <= n/4; i++) {
       if (i > 0 && i % 8 == 0) {
         debug_print("\n");
       }
       for (j = 1; j <= 4; j++) {
         if ((i*4+4-j) < n) {
           debug_print("%.2X", ptr[(i*4)+4-j]);
         }
       }
       debug_print(" ");
     }
     debug_print("\n");
   }

   <CODE ENDS>
```

Appendix C. Acknowledgements

The authors would like to thank the following persons for their valuable advice and input.

- Erick Borsboom, for assisting the lengthy review of this document;
- Jack Lloyd and Daniel Wyatt, of the Ribose RNP team, for their input and implementation;
- Paul Yang, for reviewing and proposing improvements to readability of this document;
- Markku-Juhani Olavi Saarinen, for reviewing and proposing inclusion of better examples and reference code to aid implementers, as well as for actually going through the examples to ensure their correctness.

Authors' Addresses

Ronald Henry Tse
Ribose
Suite 1111, 1 Pedder Street
Central, Hong Kong
People's Republic of China

Email: ronald.tse@ribose.com
URI: https://www.ribose.com

Wai Kit Wong
Hang Seng Management College
Hang Shin Link, Siu Lek Yuen
Shatin, Hong Kong
People's Republic of China

Email: wongwk@hsmc.edu.hk
URI: https://www.hsmc.edu.hk