Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: May 28, 2018

S. Shen
CNIC, Chinese Academy of Sciences
X. Lee
ICT, Chinese Academy of Sciences
R. Tse
Ribose
W. Wong
Hang Seng Management College
Y. Yang
BaishanCloud

November 24, 2017

The SM3 Cryptographic Hash Function
draft-oscca-cfrg-sm3-02

Abstract

This document describes the SM3 cryptographic hash algorithm published as GB/T 32905-2016 by the Organization of State Commercial Administration of China (OSCCA).

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

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 28, 2018.

Copyright Notice

Table of Contents

1. Introduction

SM3 [GBT.32905-2016] [ISO.IEC.10118-3] is a cryptographic hash algorithm published by the Organization of State Commercial Administration of China [OSCCA] as an authorized cryptographic hash algorithm for the use within China. The algorithm is published in public.

The SM3 algorithm is intended to address multiple use cases for commercial cryptography, including, but not limited to:

- the use of digital signatures and their verification;
- the generation and verification of message authenticity codes; as well as
- the generation of random numbers.

SM3 has a Merkle-Damgard construction and is similar to SHA-2 [NIST.FIPS.180-4] of the MD4 [RFC6150] family, with the addition of several strengthening features including a more complex step function and stronger message dependency than SHA-256 [SM3-Details].

SM3 produces an output hash value of 256 bits long, based on 512-bit input message blocks, on input lengths up to 2^(m) [GBT.32905-2016].

This document details the SM3 algorithm and its internal steps together with demonstrative examples.

1.1. Purpose

This document does not aim to introduce a new algorithm, but to provide a clear and open description of the SM3 algorithm in English, and also to serve as a stable reference for IETF documents that utilize this algorithm.

This document follows the updated description and structure of [GBT.32905-2016] published in 2016.

Section 1 to Section 5 of this document directly map to the corresponding sections (and numbering) of the [GBT.32905-2016] standard for convenience of the reader.

Section 6 to Section 7 of this document provides a translation of the design considerations, hardware adaptability, and cryptanalysis results of SM3 in the words of its designer, Xiaoyun Wang, given in [SM3-Details]. The cryptanalysis section has also been updated to include the latest published research on SM3.

1.2. History

The SM3 algorithm was designed by Xiaoyun Wang [WXY] et al.

It was first published by the OSCCA [OSCCA] in public in 2010 [OSCCA-SM3], then published as a China industry standard in 2012 [GMT-0004-2012], and finally published as a Chinese National Standard (GB Standard) [GBT.32905-2016] in 2016. SM3 has been standardized in [ISO.IEC.10118-3] by the International Organization for Standardization in 2017.

The latest SM3 standard [GBT.32905-2016] was proposed by the OSCCA, standardized through TC 260 of the Standardization Administration of the People's Republic of China (SAC), and was drafted by the following individuals at Tsinghua University, the China Commercial Cryptography Testing Center, the People's Liberation Army Information Engineering University, and the Data Assurance and Communication Security Research Center (DAS Center) of the Chinese Academy of Sciences:

- Xiao-Yun Wang
- Zheng Li
- Yong-Chuan Wang
- Hong-Bo Yu
- Yong-Quan Xie
- Chao Zhang
- Peng Luo
- Shu-Wang Lu

1.3. Applications

SM3 has prevalent hardware implementations, due to its being the only OSCCA-approved cryptographic hash algorithm allowed for use in China [SM3-Details].

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.

bit string

a binary string composed of 0s and 1s.

big-endian

describes the order in which data is stored in memory, where the more significant digits are stored at the lower storage addresses, the less significant digits are stored at the high storage addresses.

message

a bit string of arbitrary length. In this document, the message is the input to the hash algorithm.

hash value

the output bit string of the hash algorithm given input of a message.

word

a 32-bit quantity.

3. Symbols And Abbreviations

3.1. Operators

bitlen(S)

The length of string S in bits (e.g., bitlen(101) == 3).

S + T

addition of two 32-bit vectors S and T with a mod 2^32 bit wrap around.

S and T

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

S or T

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

S xor T

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

not(S)

bitwise "not" of a 32-bit vectors S.

a <<< i

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

S || T

String S concatenated with string T (e.g., 000 || 111 == 000111).

a <- S

Assignment operator of value S to variable a.

num2str(i, n)

The n-bit string whose base-2 interpretation is i (e.g., num2str(14,4) == 1110 and num2str(1,2) == 01).

3.2. Usage

A, B, C, D, E, F, G, H

Each a 8 word-width register.

B_i

The i-th message section

CF

The compression function.

FF_j, GG_j

Boolean functions, changes according to j.

IV

The initialization vector, used to determine the initial state of the compression function registers.

P_0

The permutation function within the compression function.

P_1

The permutation function for message expansion.

T_j

The algorithm constant, changes according to j.

m

The message.

m'

The message m after padding.

n

Number of message blocks within a message.

4. Primitives And Functions

4.1. Initialization Vector IV

IV = 7380166f 4914b2b9 172442d7 da8a0600
a96f30bc 163138aa e38dee4d b0fb0e4e

4.2. Constants T_j

When 0 <= j <= 15:

T_j = 79cc4519

When 16 <= j <= 63:

T_j = 7a879d8a

Selection of T_j is based on considerations provided in Section 6.2.6.

4.3. Boolean Functions FF_j and GG_j

When 0 <= j <= 15:

FF_j(X, Y, Z) = X xor Y xor Z
GG_j(X, Y, Z) = X xor Y xor Z

When 16 <= j <= 63:

FF_j(X, Y, Z) = (X and Y) or (X and Z) or (Y and Z)
GG_j(X, Y, Z) = (X and Y) or (not(X) and Z)

Where X, Y, Z are 32-bit words.

Note that FF_j and GG_j are identical for 0 <= j <= 15. Design considerations of these boolean functions are detailed in Section 6.2.3.

4.4. Permutation Functions P_0 and P_1

P_0(X) = X xor (X <<< 9) xor (X <<< 17)
P_1(X) = X xor (X <<< 15) xor (X <<< 23)

Where X is a 32-bit word.

Design considerations of these permutation functions are detailed in Section 6.2.2.

5. Algorithm

5.1. Overview

The SM3 cryptographic hash algorithm takes input of a message m of length l (where l < 2^64), and after padding and iterative compression, creates a hash value of 256-bits long.

Examples are provided in Appendix A.

5.2. Padding PAD

The following steps pads a message m to m', where bitlen(m') is a multiple of 512.

1. Input message m has a length of l bits.

2. Append a bit "1" to the end of the message m.

3. Append a k-bit string K, which is a series of "0"s, to the end of message m, where k is the smallest non-negative number that satisfies l + 1 + k = 448 (mod 512).

4. Append a 64-bit bit string L, where L = num2str(l, 64).

Inputs:

o m, the original message m of length l bits.

Output:

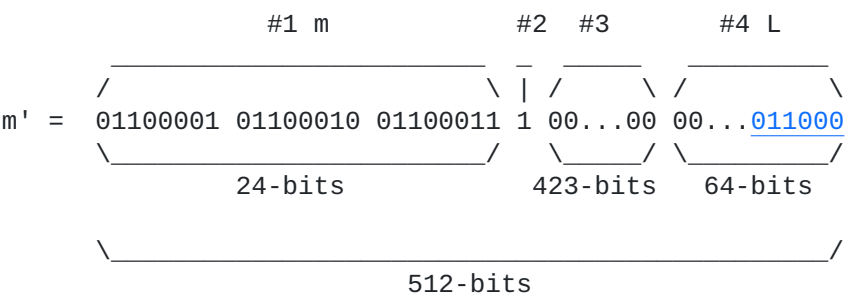
o m', the padded message of m, where bitlen(m') is a multiple of 512.

m' is defined as follows:

```
l = bitlen(m)
L = num2str(l, 64)
k = 512 - (((l mod 512) + 1 + 64) mod 512)
K = num2str(0, k)

m' = m || 1 || K || L
```

For example, given a message "01100001 01100010 01100011", its length l is 24, after padding m' will be:



5.3. Iterative Hashing

5.3.1. Iterative Compression Process

Inputs:

o m', the padded message of m, composed of n 512-bit blocks, where n = (l + k + 65) / 512

o IV, a 256-bit initialization vector

Output:

o V_n, the resulting hash value of m.

V_n is defined as follows.

```
m' = B_0 || B_1 || ... || B_{n - 1}
V_0 = IV

for i = 0 to n - 1
  E_i = ME(B_i)
  V_{i + 1} = CF(V_i, E_i)
end for

V_n
```

Where,

o CF is the compression function;

o ME is the message expansion function;

o B_i is the i-th block of the padded message m'.

5.3.2. Message Expansion Function ME

This steps expands each message block B_i into bit string E_i for the compression function CF, where E is made up of 132 words: E_i = W_0 || ... || W_67 || W'_0 || ... || W'_63.

Inputs:

o B_i, the i-th message block of the padded message m'

Output:

o E_i, the result of the message expansion function

ME(B_i) is defined as the following:

E_i is defined as follows.

```
B_i = W_0 || ... || W_15

for j = 16 to 67
  W_j = P_1(W_{j - 16} xor W_{j - 9} xor (W_{j - 3} <<< 15)) xor
          (W_{j - 13} <<< 7) xor
          W_{j - 6}
end for

for j = 0 to 63
  W'_j = W_j xor W_{j + 4}
end for

E_i = W_0 || ... || W_67 || W'_0 || ... || W'_63
```

The design considerations of ME are detailed in Section 6.3.

Selection criteria for the rotational constants 7 and 15 are provided in Section 6.2.5.

5.3.3. Compression Function CF

CF(V_i, E_i) is defined as the following function.

Inputs:

- V_i, the output value of the i-th iteration
- E_i, the expanded form of the i-th message block B_i

Variables:

- A, B, C, D, E, F, G, H, 32-bit registers
- SS1, SS2, TT1, TT2, 32-bit intermediate variables

Output:

- V_{i + 1}, the result of the compression function, where 0 <= i <= n - 1.

V_{i + 1} defined as follows.

```
A || B || C || D || E || F || G || H <- V_i
W_0 || ... || W_67 || W'_0 || ... || W'_63 <- E_i

for j = 0 to 63
  SS1 <- ((A <<< 12) + E + (T_j <<< (j mod 32))) <<< 7
  SS2 <- SS1 xor (A <<< 12)
  TT1 <- FF_j(A, B, C) + D + SS2 + W'_j
  TT2 <- GG_j(E, F, G) + H + SS1 + W_j
  D <- C
  C <- B <<< 9
  B <- A
  A <- TT1
  H <- G
  G <- F <<< 19
  F <- E
  E <- P_0(TT2)
end for

V_{i + 1} = (A || B || C || D || E || F || G || H) xor V_i
```

All 32-bit words used here are stored in big-endian format.

The design considerations of CF are detailed in Section 6.2.

5.3.4. Hash Value

The final hash value y, of 256 bits long, is given by:

y = V_n

6. Design Rationale

SM3's iterative compression function, while similar in structure to that of SHA-256, incorporates a number of novel design techniques including its 16 steps of pure exclusive-or operations, double-word message entry and the permutation function P that accelerates the avalanche effect. These techniques reduces its sensitivity to locality and increases both weak and strong collision resistance, against differential cryptanalysis, linear cryptanalysis and bit-tracing cryptanalysis techniques [SM3-Details].

The SM3 algorithm uses word addition, carry operations and a 4-stage pipeline. The P permutation is used to accelerate the avalanche effect and efficiency of the algorithm without increasing cost of hardware.

SM3 is designed to be highly efficient and widely applicable across platforms, and its operations can be easily realized in hardware on 32-bit microprocessors and 8-bit smartcards.

6.1. General Design Principles

The design of SM3 took into account of the following principles:

1. Effectively resist bit-tracing and other cryptanalysis techniques;

2. Reasonable requirements for implementation in hardware and software; and

3. Generally match or exceed performance of SHA-256 under the same conditions, while satisfying security requirements.

6.2. Compression Function Design

6.2.1. Compression Function Design Principles

The SM3 compression function is designed to have a clear structure and provide a strong avalanche effect, utilizing the following design techniques.

1. Double-word message intervention. The double-word message input is selected from the output of the message expansion algorithm. To produce the avalanche effect as early as possible, mod 2^32 arithmetic addition and the P permutations are used.

2. Each step uses message bits from the previous step for nonlinear rapid diffusion, each message bit is rapidly incorporated into the current step's diffusion and mixing.

3. Uses a mixture of different groups of operations, including modulus 2^32 addition, exclusive-or, ternary boolean functions and P permutations.

4. While satisfying the security requirements, the algorithm should be easily realized in hardware and smartcards and therefore its nonlinear operations mainly utilize boolean and additive operations.

5. Compression function parameters should facilitate the characteristics of diffusion completeness and the rapid avalanche effect.

6.2.2. Selection Of Permutation Function Parameters

The selection of permutation P_0 constants should exclude short displacement distances, bit-shifts at word-length multiples and bit-shifts of composite numbers.

Numbers 9 and 17 have been selected as shift constants having considered the security and implementability of the algorithm.

6.2.3. Selection Of Boolean Functions

Boolean functions are used to guard against bit-tracing cryptanalysis techniques, improve the nonlinearity and reduce differential image characteristics.

The selection of boolean functions should fulfill the following requirements.

1. Steps 0-15 uses pure exclusive-or operations to prevent bit-tracing.

2. Steps 16-63 use nonlinear operations to improve the algorithm's nonlinearity. At the same time, bits should be well-diffused to combine with the shift performed inside the compression function to reduce differentials between input and output.

3. The function should be a non-degenerate boolean function that is 0- and 1-balanced.

4. The boolean function must be obvious and simple to understand, as well as easy to implement.

6.2.4. Selection Of Rotational Constants R and R'

The selection of rotational constants R and R' are based on the following requirements:

1. When value x is rotated on 0-15, R . x mod 32, R' . x mod 32, R + R') . x mod 32$$ is well distributed among 0-31, making message diffusion more balanced. See Figure 1.

2. R and R' should complement the rotational constants S, S' as well as the permutation P_0 to accelerate diffusion of message bits.

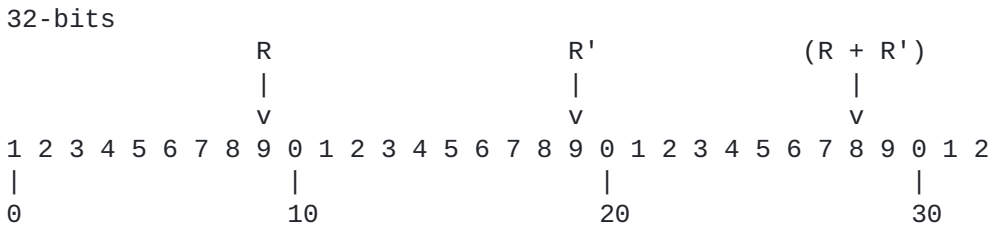


Figure 1: Selection of Rotational Constants R and R' (9 and 19)

6.2.5. Selection Of Rotational Constants S and S'

Rotational constants S and S' are used to accelerate message-bit diffusion and to increase mixture of the three inputs of the boolean functions. These constants are used in the message expansion stage.

The selection of rotational constants S and S' are based on the following requirements:

1. The absolute difference of S and S' should be around 8. S' should be a prime number, S should be a "further" odd number, to make message diffusion more balanced.

2. Needs to complement the rotational constants R and R' in accelerating diffusion of message bits.

3. The choice of S and S' should be easily implementable on 8-bit smartcards.

4. S and S' should not counteract the functionality of the permutation P_0, especially the avalanche effect.

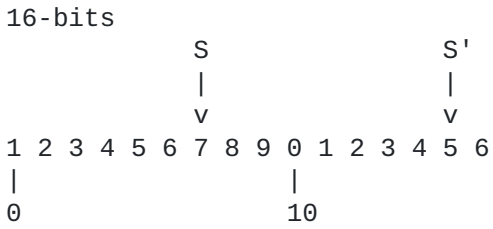


Figure 2: Selection of Rotational Constants S and S' (7 and 15)

6.2.6. Selection Of Addition Constants

Addition constants are used to provide randomness.

For mod 2^32 calculations, addition constants can reduce the linearity and probability of differential inheritance [ADDEND.DEPS].

The requirements for the addition constants are:

1. The addition constants should be 0-, 1-balanced in binary form.

2. The addition constants in binary form, should have a maximum run length of 1 and 0 of less than 5 and 4 respectively.

3. Addition constants should be easy to represent and memorize.

6.3. Message Expansion Design

Message expansion is used to expand a message block of 512 bits to 2176 bits. A better diffusion effect with minimal computation is achieved through the usage of linear feedback shift registers.

The message expansion algorithm is mainly used to enhance the correlation between message bits, and reduce the possibility of attacking the SM3 algorithm through message expansion vulnerabilities.

Requirements of the message extension algorithm are:

1. The algorithm must be entropy-preserving.

2. Linear expansion of the message to preserve correlation within the expanded message.

3. Provides a strong avalanche effect.

4. Suitable for hardware and smartcard implementations.

7. Cryptanalysis

This section provides the latest cryptanalysis results of the SM3 algorithm, and compares it with cryptanalysis of algorithms specified in [ISO.IEC.10118-3] as well as other standard hash algorithms.

7.1. Analysis

Current published cryptanalysis research mainly focuses on collision attacks, preimage attacks and distinguishing attacks.

7.1.1. Differential Analysis

Modular differential cryptanalysis [MD4-Coll] [MD5-Coll] [SHA1-Coll] is the most common method for finding collisions of hash algorithms.

It is generally described as the following steps:

1. Select a proper input difference, which decides the probability of a successful attack.

2. Search for a feasible differential path for the selected input difference.

3. Export the sufficient conditions that guarantee the feasibility of the differential path. During the search of the differential path, if conditions of chaining variables are fixed, a feasible differential path means all conditions of chaining variables that are derived from the path have no conflicts with each other.

4. Apply message modification techniques to fulfill as many sufficient conditions as possible.

Several automated searching methods for differential paths have been published in recent years [SHA1-Char] [SHA2-Char].

Based on differential characteristics of the SM3 algorithm, Mendel et al. [SM3-Coll] presented a 20-step collision attack and a 24-step freestart collision attack against SM3 at CT-RSA 2013.

7.1.2. Preimage Attacks

Preimage attacks against hash algorithms with a Merkle-Damgard construction have been mainly the meet-in-the-middle attack [NBS-Crypt] [MD4-Preimage] and its improved variants, such as the differential meet-in-the-middle technique [SHA1-Pre].

While searching for preimages, the pseudo preimage of a single message block has to be first found, and then the pseudo preimage is converted to a preimage of multiple blocks [HAC].

The steps of finding a pseudo preimage can be generally described as:

1. Select a proper independent message word (or bit), note as independent message word 1 and independent message word 2 and split the compression function into three parts, the independent part 1, the independent part 2 and the match part base on the independent message. The independent message word 1 and the independent part 2 are independent from each other, as well as the independent message word 2 and the independent part 1.

2. Randomly set other messages other than the independent message word 1 and 2, and also the chaining variables of the independent part 1 and 2.

3. Calculate list L_1 by independent message word 1 and independent part 1. Calculate list L_2 by independent message word 2 and independent part 2.

4. Search for a collision of L_1 and L_2, the corresponding initial value and message of this collision will be a pseudo preimage.

The biclique attack is an initial structure for creating meet-in-the-middle attacks [SHA2-Pre]. By using bicliques, Zou et al. [SM3-Pre] at ICISC 2011, presented a preimage attack on SM3 from step 1 to step 28, and a 30-step preimage attack that starts from the middle.

In 2012, Wang and Shen [SM3-Pre2] mounted a differential biclique attack to give a 29 and 30 step preimage attack against SM3, as well as a 31 and 32 step pseudo preimage attack against SM3. These results start from step 1.

7.1.3. Distinguishing Attacks

The boomerang attack is the main distinguishing attack used against SM3.

The boomerang attack uses chaining variables from one or multiple steps, to form a long differential path by connecting two short differential paths, and then constructing a quartet that can fulfill the input and output differentials.

The process is generally described as the following steps:

1. Select a proper message differential and construct the short differential paths. The message differential should be selected that the sufficient conditions appear around the conjunction position.

2. Test sufficient conditions that are around the conjunction position to see if they conflict.

3. Randomly select chaining variables at the conjunction position and then apply message modification techniques to allow the modified message to fulfill as many sufficient conditions as possible.

4. Start from the conjunction position, construct corresponding differential paths toward each side, to derive corresponding input and output differentials.

At SAC 2012, Kircanski et al. [SM3-Boomerang] presented 32-step to 35-step boomerang distinguishing attack against SM3 algorithm along

with the instances of 32-step and 33-step attack. They also utilized the shifting characteristic of SM3 algorithm to replace all non-linear operations with XOR operations to get the SM3-XOR characteristic.

In 2014, Bai et al. [SM3-Boomerang2] improved the boomerang attack against SM3 with 34 to 37 step attacks, and presented instances of that attack at 34 and 35 steps.

The best cryptanalysis results of the SM3 algorithm are shown in Table 1 as of publication of this document.

Attack Type	Target	Steps	Complexity	Reference
Collision	HF	20	Practical	[SM3-Coll]
Freestart Collision	CF	24	Practical	[SM3-Coll]
Preimage	HF	28	2^241.5	[SM3-Pre]
Preimage	HF	30	2^249	[SM3-Pre]
Preimage	HF	29	2^245	[SM3-Pre2]
Preimage	HF	30	2^251.1	[SM3-Pre2]
Pseudo-Preimage	CF	31	2^245	[SM3-Pre2]
Pseudo-Preimage	CF	32	2^251.1	[SM3-Pre2]
Boomerang	CF	33	Practical	[SM3-Boomerang]
Boomerang	CF	35	2^117.1	[SM3-Boomerang]
Boomerang	CF	35	Practical	[SM3-Boomerang2]
Boomerang	CF	37	2^192	[SM3-Boomerang2]

Table 1: SM3 Cryptanalysis Summary (CF: Compression Function, HF: Hash Function)

7.2. Comparison with Other Algorithms

The results of SM3 algorithm compares with other hash algorithms as SHA-1 [NIST.FIPS.180-1], SHA-2 [NIST.FIPS.180-4] [NIST.FIPS.180-2], RIPEMD-128 [RIPEMD], RIPEMD-160 [RIPEMD], Whirlpool [WHIRLPOOL], Stribog [GOSTR.34.11.2012] and SHA-3 [NIST.FIPS.202] are shown in Table 2.

Algorithm	Attack Type	Steps / Rounds	%	References
SM3	Collision	20	31	[SM3-Coll]
SM3	Preimage	30	47	[SM3-Pre]
				[SM3-Pre2]
SM3	Distinguisher	37	58	[SM3-Boomerang2]
SHA-1	Collision	80	100	[SHA1-Coll]
				[SHA1-Coll2]
				[SHA1-Coll3]
SHA-1	Preimage	62	77.5	[SHA1-HDPre]
RIPEMD-128	Collision	40	62.5	[RIPE128-Coll]
RIPEMD-128	Preimage	36	62.5	[RIPE128-Pre]
RIPEMD-128	Distinguisher	64	100	[RIPE128-Crypt]
RIPEMD-160	Preimage	34	53.12	[RIPE160-Pre]
RIPEMD-160	Distinguisher	51	79.68	[RIPE-Dist]
SHA-256	Collision	31	48.4	[SHA256-Coll]
SHA-256	Preimage	45	70.3	[SHA2-Pre]
SHA-256	Distinguisher	47	73.4	[SHA256-Diff]
Whirlpool	Collision	8	80	[WP-PC]
Whirlpool	Preimage	6	60	[WP-PC]
Whirlpool	Distinguisher	10	100	[WP-Rebound]
Stribog-256	Collision	6.5	54.2	[ST-Pre]
Stribog-512	Collision	7.5	62.5	[ST-Pre]
Stribog-512	Preimage	6	50	[ST-Pre]
Stribog-512	Distinguisher	6	50	[ST-Boom]
SHA3-224	Collision	5	20.8	[SHA3-SLin]
SHA3-256	Collision	5	20.8	[SHA3-Coll]
SHA3-256	Preimage	4	16.7	[SHA3-Rot]
SHA3-512	Collision	3	12.5	[SHA3-Coll]
SHAKE-128	Collision	5	20.8	[SHA3-Coll2]
Keccak-f	Distinguisher	24	100	[KEKKAC-ZSD]

Table 2: SM3 Cryptanalysis Comparison

Table 2 indicates:

- Collision attacks: the attack percentage of SM3 is slightly higher than SHA-3, lower than the other compared algorithms, and the lowest among MD-SHA-like algorithms at 31% of steps.

- Preimage attacks: the attack percentage of SM3 is slightly higher than SHA-3, lower than the other compared algorithms, and the lowest among MD-SHA-like algorithms at 47% of steps.

- Distinguisher attacks: the attack percentage of SM3 is lower than all compared algorithms, with only 58% of steps distinguished.

These results reflect that the SM3 algorithm is highly resistant.

8. Object Identifier

The Object Identifier for SM3 is identified through these OIDs.

8.1. GM/T OID

- "1.2.156.10197.1.401" for "Hash Algorithm: SM3 Algorithm" [GMT-0004-2012].

- "1.2.156.10197.1.401.1" for "Hash Algorithm: SM3 Algorithm used without secret key" [GMT-0004-2012].

- "1.2.156.10197.1.401.2" for "Hash Algorithm: SM3 Algorithm used with secret key" [GMT-0004-2012].

8.2. ISO OID

"1.0.10118.3.0.65" for "id-dhf-SM3" [ISO.IEC.10118-3], described below.

- "is10118-3" {iso(1) standard(0) hash-functions(10118) part3(3)}

- "id-dhf" { is10118-3 algorithm(0) }

- "id-dhf-SM3" { id-dhf sm3 (65) }

8.3. OID For Digital Signatures

- "1.2.156.10197.1.501" for "Digital Signature: Based on SM2 and SM3"

- "1.2.156.10197.1.504" for "Digital Signature: Based on RSA and SM3"

9. Security Considerations

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

- SM3 [GBT.32905-2016] is a cryptographic hash algorithm published by the OSCCA [OSCCA]. No formal proof of security is provided. The security properties of SM3 are under public study. There are

no known feasible attacks against the SM3 algorithm at the time this document is published.

- SM3 is a hash function that generates a 256-bit hash value. It is considered as an alternative to SHA-256 [RFC6234].

10. IANA Considerations

This document does not require any action by IANA.

11. References

11.1. Normative References

[GBT.32905-2016]
Standardization Administration of the People's Republic of China, "GB/T 32905-2016: Information security techniques -- SM3 cryptographic hash algorithm", August 2016, <www.gb688.cn/bzgk/gb/newGbInfo?hcno=45B1A67F20F3BF339211C391E9278F5E>.

[ISO.IEC.10118-3]
International Organization for Standardization, "ISO/IEC FDIS 10118-3 -- Information technology -- Security techniques -- Hash-functions -- Part 3: Dedicated hash-functions", September 2017, <https://www.iso.org/standard/67116.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>.

11.2. Informative References

[ADDEND.DEPS]
Miyano, H., "Addend Dependency of Differential/Linear Probability of Addition", January 1998, <https://search.ieice.org/bin/summary.php?id=e81-a_1_106>.

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

[GBT.32918.2-2016]
Standardization Administration of the People's Republic of China, "GB/T 32918.2-2016 Information Security Technology -- Public Key Cryptographic Algorithm SM2 Based On Elliptic Curves -- Part 2: Digital Signature Algorithm", August 2016, <http://www.gb688.cn/bzgk/gb/newGbInfo?hcno=6F1FAEB62F9668F25F38E0BF0291D4AC>.

[GBT.32918.3-2016]
Standardization Administration of the People's Republic of China, "GB/T 32918.3-2016 Information Security Technology -- Public Key Cryptographic Algorithm SM2 Based On Elliptic Curves -- Part 3: Key Exchange", August 2016, <http://www.gb688.cn/bzgk/gb/newGbInfo?hcno=66A89DD6DA64F49C49456B757BA0624F>.

[GBT.32918.4-2016]
Standardization Administration of the People's Republic of China, "GB/T 32918.4-2016 Information Security Technology -- Public Key Cryptographic Algorithm SM2 Based On Elliptic Curves -- Part 4: Public Key Encryption Algorithm", August 2016, <http://www.gb688.cn/bzgk/gb/newGbInfo?hcno=370AF152CB5CA4A377EB4D1B21DECAE0>.

[GMSSL]
Peking University, "The GmSSL Project", October 2017, <https://www.gmssl.org>.

[GMT-0004-2012]
Organization of State Commercial Administration of China, "GM/T 0004-2012: SM3 Cryptographic Hash Algorithm", March 2012, <http://www.oscca.gov.cn/Column/Column_32.htm>.

[GOSTR.34.11.2012]
Federal Agency on Technical Regulation And Metrology, "GOST R 34.11-2012: Information technology -- Cryptographic Data Security -- Hash-function", August 2012, <https://tc26.ru/en/research/polozhenie/GOST-R-34-11-2012.php>.

[HAC]
Menezes, A., van Oorschot, P., and S. Vanstone, "Handbook of Applied Cryptography", 2011, <http://cacr.uwaterloo.ca/hac/>.

[KEKKAC-ZSD]
Duan, M. and X. Lau, "Improved zero-sum distinguisher for full round Keccak-f permutation", 2012, <https://doi.org/10.1007/s11434-011-4909-x>.

[MD4-Coll]
Wang, X., Feng, D., Lau, X., and H. Yu, "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD", 2004, <https://eprint.iacr.org/2004/199>.

[MD4-Preimage]
Aoki, K. and Y. Sasaki, "Preimage attacks on one-block MD4, 63-step MD5 and more", 2008, <https://doi.org/10.1007/978-3-642-04159-4_7>.

[MD5-Coll]
Wang, X. and H. Yu, "How to Break MD5 and Other Hash Functions", 2005, <https://doi.org/10.1007/11426639_2>.

[NBS-Crypt]
Diffie, W. and M. Hellman, "Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard", 1977, <https://doi.org/10.1109/C-M.1977.217750>.

[NIST.FIPS.180-1]
Barker, E., "NIST Federal Information Processing Standard 180-1: Secure Hash Standard (SHS)", April 1995, <https://www.nist.gov/publications/secure-hash-standard-shs-2?pub_id=917977>.

[NIST.FIPS.180-2]
Barker, E., "NIST Federal Information Processing Standard 180-2: Secure Hash Standard (SHS)", August 2002, <https://csrc.nist.gov/csrc/media/publications/fips/180/2/archive/2002-08-01/documents/fips180-2.pdf>.

[NIST.FIPS.180-4]
National Institute of Standards and Technology, "FIPS 180-4 Secure Hash Standard (SHS)", August 2015, <http://dx.doi.org/10.6028/NIST.FIPS.180-4>.

[NIST.FIPS.202]
National Institute of Standards and Technology, "NIST Federal Information Processing Standard 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <https://doi.org/10.6028/NIST.FIPS.202>.

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

[OSCCA] Organization of State Commercial Administration of China, "Organization of State Commercial Administration of China", May 2017, <http://www.oscca.gov.cn>.

[OSCCA-SM3] Organization of State Commercial Administration of China, "SM3 Cryptographic Hash Algorithm", December 2010, <http://www.oscca.gov.cn/UpFile/20101222141857786.pdf>.

[RFC6150] Turner, S. and L. Chen, "MD4 to Historic Status", RFC 6150, DOI 10.17487/RFC6150, March 2011, <https://www.rfc-editor.org/info/rfc6150>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RIPE-Dist] Sasaki, Y. and L. Wang, "Distinguishers beyond Three Rounds of the RIPEMD-128/-160 Compression Functions", 2012, <https://doi.org/10.1007/978-3-642-31284-7_17>.

[RIPE128-Coll] Wang, G., "Practical collision attack on 40-step RIPEMD-128", 2014, <https://doi.org/10.1007/978-3-319-04852-9_23>.

[RIPE128-Crypt] Landelle, F. and T. Peyrin, "Cryptanalysis Of Full RIPEMD-128", 2014, <https://doi.org/10.1007/978-3-642-38348-9_14>.

[RIPE128-Pre] Wang, L., Sasaki, Y., Komatsubara, W., Ohta, K., and K. Sakiyama, "(Second) Preimage Attacks on Step-Reduced RIPEMD/RIPEMD-128 with a New Local-Collision Approach", 2011, <https://doi.org/10.1007/978-3-642-19074-2_14>.

[RIPE160-Pre] Wang, G. and Y. Shen, "(Pseudo-) Preimage Attacks on Step-Reduced HAS-160 and RIPEMD-160", 2014, <https://doi.org/10.1007/978-3-319-13257-0_6>.

[RIPEMD] Dobbertin, H., Bosselaers, A., and B. Preneel, "RIPEMD-160: A Strengthened Version of RIPEMD", 1996, <http://www.esat.kuleuven.be/~bosselae/ripemd160.html>.

[SHA1-Char]
De Cannire, C. and C. Rechberger, "Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family", 2012, <https://doi.org/10.1007/11935230_1>.

[SHA1-Coll]
Wang, X., Yin, Y., and H. Yu, "Finding collisions in the full SHA-1", 2005, <https://doi.org/10.1007/11535218_2>.

[SHA1-Coll2]
Wang, X., Yao, A., and F. Yao, "Cryptanalysis on SHA-1", 2005, <https://csrc.nist.gov/csrc/media/events/first-cryptographic-hash-workshop/documents/wang_sha1-new-result.pdf>.

[SHA1-Coll3]
Stevens, M., "New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis", 2013, <https://doi.org/10.1007/978-3-642-38348-9_15>.

[SHA1-HDPre]
Espitau, T., Fouque, P-A., and P. Karpman, "Higher-Order Differential Meet-in-the-middle Preimage Attacks on SHA-1 and BLAKE", 2015, <https://doi.org/10.1007/978-3-662-47989-6_33>.

[SHA1-Pre]
Khovratovich, D. and S. Knellwolf, "New Preimage Attacks against Reduced SHA-1", 2012, <https://doi.org/10.1007/978-3-642-32009-5_22>.

[SHA2-Char]
Mendel, F., Nad, T., and M. Schlaffer, "Finding SHA-2 Characteristics: Searching through a Minefield of Contradictions", 2011, <https://doi.org/10.1007/978-3-642-25385-0_16>.

[SHA2-Pre]
Khovratovich, D., Rechberger, C., and A. Savelieva, "Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family", 2012, <https://doi.org/10.1007/978-3-642-34047-5_15>.

[SHA256-Coll]
Mendel, F., Nad, T., and M. Schlaffer, "Improving Local Collisions: New Attacks on Reduced SHA-256", 2013, <https://doi.org/10.1007/978-3-642-38348-9_16>.

[SHA256-Diff]
Biryukov, A., Lamberger, M., Mendel, F., and I. Nikolic, "Second-Order Differential Collisions for Reduced SHA-256", 2011, <https://doi.org/10.1007/978-3-642-25385-0_15>.

[SHA3-Coll]
Dinur, I., Dunkelman, O., and A. Shamir, "Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials", 2014, <https://doi.org/10.1007/978-3-662-43933-3_12>.

[SHA3-Coll2]
Qiao, K., Song, L., Liu, M., and J. Guo, "New Collision Attacks on Round-Reduced Keccak", 2017, <https://doi.org/10.1007/978-3-319-56617-7_8>.

[SHA3-Rot]
Morawiecki, P., Pieprzyk, J., and M. Srebrny, "Rotational Cryptanalysis of Round-Reduced Keccak", 2013, <https://doi.org/10.1007/978-3-662-43933-3_13>.

[SHA3-SLin]
Song, L., Liao, G., and J. Guo, "Non-Full Sbox Linearization: Applications to Collision Attacks on Round-Reduced Keccak", 2017, <https://doi.org/10.1007/978-3-319-63715-0_15>.

[SM3-Boomerang]
Kircanski, A., Wang, G., Shen, Y., and A. Youssef, "Boomerang and slide-rotational analysis of the SM3 hash function", 2013, <https://doi.org/10.1007/978-3-642-35999-6_20>.

[SM3-Boomerang2]
Bai, D., Yu, H., Wang, G., and X. Wang, "Improved Boomerang Attacks on Round-Reduced SM3 and Keyed Permutation of BLAKE-256", April 2015, <https://doi.org/10.1049/iet-ifs.2013.0380>.

[SM3-Coll]
Mendel, F., Nad, T., and M. Schlaffer, "Finding collisions for round-reduced SM3", 2013, <https://doi.org/10.1007/978-3-642-36095-4_12>.

[SM3-Details]
Wang, X. and H. Yu, "SM3 Cryptographic Hash Algorithm", October 2016, <http://ris.sic.gov.cn/EN/Y2016/V2/I11/983>.

[SM3-Pre] Zou, J., Wu, W., Wu, S., Su, B., and L. Dong, "Preimage attacks on step-reduced SM3 hash function", 2012, <https://doi.org/10.1007/978-3-642-31912-9_25>.

[SM3-Pre2] Zou, G. and Y. Shen, "Preimage and Pseudo-Collision Attacks on Step-Reduced SM3 Hash Function", 2013, <https://doi.org/10.1016/j.ipl.2013.02.006>.

[ST-Boom] AlTawy, R. and A. Youssef, "Preimage Attacks on Reduced-Round Stribog", 2014, <https://doi.org/10.1007/978-3-319-06734-6_7>.

[ST-Pre] Ma, B., Li, B., Hao, R., and X. Li, "Improved Cryptanalysis on Reduced-Round GOST and Whirlpool Hash Function", 2014, <https://doi.org/10.1007/978-3-319-07536-5_18>.

[WHIRLPOOL] Rijmen, V. and P. Barreto, "The WHIRLPOOL Hash Function", 2001, <http://www.larc.usp.br/~pbarreto/WhirlpoolPage.html>.

[WP-PC] Sasaki, Y., Wang, L., Wu, S., and W. Wu, "Investigating Fundamental Security Requirements on Whirlpool: Improved Preimage and Collision Attacks", 2012, <https://doi.org/10.1007/978-3-642-34961-4_34>.

[WP-Rebound] Lamberger, M., Mendel, F., Schlaffer, M., Rechberger, C., and V. Rijmen, "The Rebound Attack and Subspace Distinguishers: Application to Whirlpool", 2011, <https://doi.org/10.1007/s00145-013-9166-5>.

[WXY] Wang, X., "Xiaoyun Wang -- Institute of Advanced Study -- Tsinghua University", October 2017, <http://www.tsinghua.edu.cn/publish/casen/1695/2010/20101224093253705266640/20101224093253705266640_.html>.

Appendix A. Example Calculations

A.1. Example 1, From GB/T 32905-2016

This is example 1 provided by [GBT.32905-2016] to demonstrate hashing of a plaintext that requires padding.

A.1.1. Hexadecimal Input Message m

The input "abc" is represented in hexadecimal form as "616263".

A.1.2. Padded Message m'

The message after padding is shown below.

```
61626380 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000018
```

A.1.3. Message After Expansion ME(m')

The message after expansion is shown below.

W_0 W_1 ... W_67:

```
61626380 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000018
9092e200 00000000 000c0606 719c70ed 00000000 8001801f 939f7da9 00000000
2c6fa1f9 adaaef14 00000000 0001801e 9a965f89 49710048 23ce86a1 b2d12f1b
e1dae338 f8061807 055d68be 86cfd481 1f447d83 d9023dbf 185898e0 e0061807
050df55c cde0104c a5b9c955 a7df0184 6e46cd08 e3babdf8 70caa422 0353af50
a92dbca1 5f33cfd2 e16f6e89 f70fe941 ca5462dc 85a90152 76af6296 c922bdb2
68378cf5 97585344 09008723 86faee74 2ab908b0 4a64bc50 864e6e08 f07e6590
325c8f78 accb8011 e11db9dd b99c0545
```

W'_0 W'_1 ... W'_63:

```
61626380 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000018 9092e200 00000000 000c0606 719c70f5
9092e200 8001801f 93937baf 719c70ed 2c6fa1f9 2dab6f0b 939f7da9 0001801e
b6f9fe70 e4dbef5c 23ce86a1 b2d0af05 7b4cbcb1 b177184f 2693ee1f 341efb9a
fe9e9ebb 210425b8 1d05f05e 66c9cc86 1a4988df 14e22df3 bde151b5 47d91983
6b4b3854 2e5aadb4 d5736d77 a48caed4 c76b71a9 bc89722a 91a5caab f45c4611
6379de7d da9ace80 97c00c1f 3e2d54f3 a263ee29 12f15216 7fafe5b5 4fd853c6
428e8445 dd3cef14 8f4ee92b 76848be4 18e587c8 e6af3c41 6753d7d5 49e260d5
```

A.1.4. Intermediate Values During Iterative Compression

```
   A        B        C        D        E        F        G        H

~~~~~~~~~~~~~~~~~~~~~~~~~~~ initial value ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
7380166f 4914b2b9 172442d7 da8a0600 a96f30bc 163138aa e38dee4d b0fb0e4e

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 0 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
b9edc12b 7380166f 29657292 172442d7 b2ad29f4 a96f30bc c550b189 e38dee4d
ea52428c b9edc12b 002cdee7 29657292 ac353a23 b2ad29f4 85e54b79 c550b189
```

```
609f2850 ea52428c db825773 002cdee7 d33ad5fb ac353a23 4fa59569 85e54b79
35037e59 609f2850 a48519d4 db825773 b8204b5f d33ad5fb d11d61a9 4fa59569
1f995766 35037e59 3e50a0c1 a48519d4 8ad212ea b8204b5f afde99d6 d11d61a9
374a0ca7 1f995766 06fcb26a 3e50a0c1 acf0f639 8ad212ea 5afdc102 afde99d6
33130100 374a0ca7 32aecc3f 06fcb26a 3391ec8a acf0f639 97545690 5afdc102
1022ac97 33130100 94194e6e 32aecc3f 367250a1 3391ec8a b1cd6787 97545690
d47caf4c 1022ac97 26020066 94194e6e 6ad473a4 367250a1 64519c8f b1cd6787
59c2744b d47caf4c 45592e20 26020066 c6a3ceae 6ad473a4 8509b392 64519c8f

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 10 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
481ba2a0 59c2744b f95e99a8 45592e20 02afb727 c6a3ceae 9d2356a3 8509b392
694a3d09 481ba2a0 84e896b3 f95e99a8 9dd1b58c 02afb727 7576351e 9d2356a3
89cbcd58 694a3d09 37454090 84e896b3 6370db62 9dd1b58c b938157d 7576351e
24c95abc 89cbcd58 947a12d2 37454090 1a4a2554 6370db62 ac64ee8d b938157d
7c529778 24c95abc 979ab113 947a12d2 3ee95933 1a4a2554 db131b86 ac64ee8d
34d1691e 7c529778 92b57849 979ab113 61f99646 3ee95933 2aa0d251 db131b86
796afab1 34d1691e a52ef0f8 92b57849 067550f5 61f99646 c999f74a 2aa0d251
7d27cc0e 796afab1 a2d23c69 a52ef0f8 b3c8669b 067550f5 b2330fcc c999f74a
d7820ad1 7d27cc0e d5f562f2 a2d23c69 575c37d8 b3c8669b 87a833aa b2330fcc
f84fd372 d7820ad1 4f981cfa d5f562f2 a5dceaf1 575c37d8 34dd9e43 87a833aa

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 20 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
02c57896 f84fd372 0415a3af 4f981cfa 74576681 a5dceaf1 bec2bae1 34dd9e43
4d0c2fcd 02c57896 9fa6e5f0 0415a3af 576f1d09 74576681 578d2ee7 bec2bae1
eeeec41a 4d0c2fcd 8af12c05 9fa6e5f0 b5523911 576f1d09 340ba2bb 578d2ee7
f368da78 eeeec41a 185f9a9a 8af12c05 6a879032 b5523911 e84abb78 340ba2bb
15ce1286 f368da78 dd8835dd 185f9a9a 62063354 6a879032 c88daa91 e84abb78
c3fd31c2 15ce1286 d1b4f1e6 dd8835dd 4db58f43 62063354 8193543c c88daa91
6243be5e c3fd31c2 9c250c2b d1b4f1e6 131152fe 4db58f43 9aa31031 8193543c
a549beaa 6243be5e fa638587 9c250c2b cf65e309 131152fe 7a1a6dac 9aa31031
e11eb847 a549beaa 877cbcc4 fa638587 e5b64e96 cf65e309 97f0988a 7a1a6dac
ff9bac9d e11eb847 937d554a 877cbcc4 9811b46d e5b64e96 184e7b2f 97f0988a

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 30 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
a5a4a2b3 ff9bac9d 3d708fc2 937d554a e92df4ea 9811b46d 74b72db2 184e7b2f
89a13e59 a5a4a2b3 37593bff 3d708fc2 0a1ff572 e92df4ea a36cc08d 74b72db2
3720bd4e 89a13e59 4945674b 37593bff cf7d1683 0a1ff572 a757496f a36cc08d
9ccd089c 3720bd4e 427cb313 4945674b da8c835f cf7d1683 ab9050ff a757496f
c7a0744d 9ccd089c 417a9c6e 427cb313 0958ff1b da8c835f b41e7be8 ab9050ff
d955c3ed c7a0744d 9a113939 417a9c6e c533f0ff 0958ff1b 1afed464 b41e7be8
e142d72b d955c3ed 40e89b8f 9a113939 d4509586 c533f0ff f8d84ac7 1afed464
e7250598 e142d72b ab87dbb2 40e89b8f c7f93fd3 d4509586 87fe299f f8d84ac7
2f13c4ad e7250598 85ae57c2 ab87dbb2 1a6cabc9 c7f93fd3 ac36a284 87fe299f
19f363f9 2f13c4ad 4a0b31ce 85ae57c2 c302badb 1a6cabc9 fe9e3fc9 ac36a284

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 40 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
55e1dde2 19f363f9 27895a5e 4a0b31ce 459daccf c302badb 5e48d365 fe9e3fc9
d4f4efe3 55e1dde2 e6c7f233 27895a5e 5cfba85a 459daccf d6de1815 5e48d365
```

```
48dcbc62 d4f4efe3 c3bbc4ab e6c7f233 6f49c7bb 5cfba85a 667a2ced d6de1815
8237b8a0 48dcbc62 e9dfc7a9 c3bbc4ab d89d2711 6f49c7bb 42d2e7dd 667a2ced
d8685939 8237b8a0 b978c491 e9dfc7a9 8ee87df5 d89d2711 3ddb7a4e 42d2e7dd
d2090a86 d8685939 6f714104 b978c491 2e533625 8ee87df5 388ec4e9 3ddb7a4e
e51076b3 d2090a86 d0b273b0 6f714104 d9f89e61 2e533625 efac7743 388ec4e9
47c5be50 e51076b3 12150da4 d0b273b0 3567734e d9f89e61 b1297299 efac7743
abddbdc8 47c5be50 20ed67ca 12150da4 3dfcdd11 3567734e f30ecfc4 b1297299
bd708003 abddbdc8 8b7ca08f 20ed67ca 93494bc0 3dfcdd11 9a71ab3b f30ecfc4

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 50 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
15e2f5d3 bd708003 bb7b9157 8b7ca08f c3956c3f 93494bc0 e889efe6 9a71ab3b
13826486 15e2f5d3 e100077a bb7b9157 cd09a51c c3956c3f 5e049a4a e889efe6
4a00ed2f 13826486 c5eba62b e100077a 0741f675 cd09a51c 61fe1cab 5e049a4a
f4412e82 4a00ed2f 04c90c27 c5eba62b 7429807c 0741f675 28e6684d 61fe1cab
549db4b7 f4412e82 01da5e94 04c90c27 f6bc15ed 7429807c b3a83a0f 28e6684d
22a79585 549db4b7 825d05e8 01da5e94 9d4db19a f6bc15ed 03e3a14c b3a83a0f
30245b78 22a79585 3b696ea9 825d05e8 f6804c82 9d4db19a af6fb5e0 03e3a14c
6598314f 30245b78 4f2b0a45 3b696ea9 f522adb2 f6804c82 8cd4ea6d af6fb5e0
c3d629a9 6598314f 48b6f060 4f2b0a45 14fb0764 f522adb2 6417b402 8cd4ea6d
ddb0a26a c3d629a9 30629ecb 48b6f060 589f7d5c 14fb0764 6d97a915 6417b402

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 60 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
71034d71 ddb0a26a ac535387 30629ecb 14d5c7f6 589f7d5c 3b20a7d8 6d97a915
5e636b4b 71034d71 6144d5bb ac535387 09ccd95e 14d5c7f6 eae2c4fb 3b20a7d8
2bfa5f60 5e636b4b 069ae2e2 6144d5bb 4ac3cf08 09ccd95e 3fb0a6ae eae2c4fb
1547e69b 2bfa5f60 c6d696bc 069ae2e2 e808f43b 4ac3cf08 caf04e66 3fb0a6ae
```

A.1.5. Hash Value

```
66c7f0f4 62eeedd9 d1f2d46b dc10e4e2 4167c487 5cf2f7a2 297da02b 8f4ba8e0
```

A.2. Example 2, From GB/T 32905-2016

This is example 2 provided by [GBT.32905-2016] to demonstrate hashing of a 512-bit plaintext.

A.2.1. 512-bit Input Message

```
61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364
61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364
```

A.2.2. Padded Message

The message after padding is shown below.

```
61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364
61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364
80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000200
```

A.2.2.1. Message Block 1

A.2.2.1.1. Expanded Message

W_0 W_1 ... W_67:

```
61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364
61626364 61626364 61626364 61626364 61626364 61626364 61626364 61626364
a121a024 a121a024 a121a024 6061e0e5 6061e0e5 6061e0e5 a002e345 a002e345
a002e345 49c969ed 49c969ed 49c969ed 85ae5679 a44ff619 a44ff619 694b6244
e8c8e0c4 e8c8e0c4 240e103e 346e603e 346e603e 9a517ab5 8a01aa25 8a01aa25
0607191c 25f8a37a d528936a 89fbd8ae 00606206 10501256 7cff7ef9 3c78b9f9
cc2b8a69 9f03f169 df45be20 9ec5bee1 0a212906 49ff72c0 46717241 67e09a19
6efaa333 2ebae676 3475c386 201dcff6 2f18fccf 2c5f2b5c a80b9f38 bc139f34
c47f18a7 a25ce71d 42743705 51baf619
```

W'_0 W'_1 ... W'_63:

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 c043c340 c043c340 c043c340 01038381
c14040c1 c14040c1 01234361 c06303a0 c06303a0 29a88908 e9cb8aa8 e9cb8aa8
25acb53c ed869ff4 ed869ff4 20820ba9 6d66b6bd 4c8716dd 8041e627 5d25027a
dca680fa 72999a71 ae0fba1b be6fca1b 32697922 bfa9d9cf 5f29394f 03fa728b
06677b1a 35a8b12c a9d7ed93 b5836157 cc4be86f 8f53e33f a3bac0d9 a2bd0718
c60aa36f d6fc83a9 9934cc61 f92524f8 64db8a35 674594b6 7204b1c7 47fd55ef
41e25ffc 02e5cd2a 9c7e5cbe 9c0e50c2 eb67e468 8e03cc41 ea7fa83d eda9692d
```

A.2.2.1.2. Intermediate Values During Iterative Compression

```
   A        B        C        D        E        F        G        H
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 0 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
7380166f 4914b2b9 172442d7 da8a0600 a96f30bc 163138aa e38dee4d b0fb0e4e
588b5dab 7380166f 29657292 172442d7 b2e561d0 a96f30bc c550b189 e38dee4d
b31cecd3 588b5dab 002cdee7 29657292 887cdf53 b2e561d0 85e54b79 c550b189
087b31df b31cecd3 16bb56b1 002cdee7 5234344f 887cdf53 0e85972b 85e54b79
17448b12 087b31df 39d9a766 16bb56b1 16372ca6 5234344f fa9c43e6 0e85972b
dca06de5 17448b12 f663be10 39d9a766 f7bc113c 16372ca6 a27a91a1 fa9c43e6
8eb847a3 dca06de5 8916242e f663be10 9fe64fb1 f7bc113c 6530b1b9 a27a91a1
0e0f1218 8eb847a3 40dbcbb9 8916242e 57e5fc4e 9fe64fb1 89e7bde0 6530b1b9
ada83827 0e0f1218 708f471d 40dbcbb9 55eb8591 57e5fc4e 7d8cff32 89e7bde0
6e12c163 ada83827 1e24301c 708f471d c26a14b8 55eb8591 e272bf2f 7d8cff32
f7578117 6e12c163 50704f5b 1e24301c 3433dd28 c26a14b8 2c8aaf5c e272bf2f
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 10 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
bc497c66 f7578117 2582c6dc 50704f5b 4f85c749 3433dd28 a5c61350 2c8aaf5c
ecc59168 bc497c66 af022fee 2582c6dc 8ce5ee61 4f85c749 e941a19e a5c61350
63723715 ecc59168 92f8cd78 af022fee 38e2aa27 8ce5ee61 3a4a7c2e e941a19e
e57bfbf8 63723715 8b22d1d9 92f8cd78 542318e7 38e2aa27 730c672f 3a4a7c2e
8ba504b1 e57bfbf8 e46e2ac6 8b22d1d9 a8c73777 542318e7 5139c715 730c672f
b6a4be20 8ba504b1 f7f7f1ca e46e2ac6 8ae4d7a0 a8c73777 c73aa118 5139c715
c0a0e3f7 b6a4be20 4a096317 f7f7f1ca f671e12a 8ae4d7a0 bbbd4639 c73aa118
68ef7357 c0a0e3f7 497c416d 4a096317 673f9d46 f671e12a bd045726 bbbd4639
4c6499d3 68ef7357 41c7ef81 497c416d f01924a3 673f9d46 0957b38f bd045726
9f532735 4c6499d3 dee6aed1 41c7ef81 71c6ef02 f01924a3 ea3339fc 0957b38f

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 20 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
231d84bd 9f532735 c933a698 dee6aed1 108149de 71c6ef02 251f80c9 ea3339fc
6a203212 231d84bd a64e6b3e c933a698 90c31af9 108149de 78138e37 251f80c9
175c3b57 6a203212 3b097a46 a64e6b3e 508f82d2 90c31af9 4ef0840a 78138e37
cdcbabd5 175c3b57 406424d4 3b097a46 b5a2f2fb 508f82d2 d7cc8618 4ef0840a
7dd941f8 cdcbabd5 b876ae2e 406424d4 a541cb9b b5a2f2fb 1692847c d7cc8618
eaf54f3e 7dd941f8 9757ab9b b876ae2e 912d4e17 a541cb9b 97ddad17 1692847c
f7310a83 eaf54f3e b283f0fb 9757ab9b b43da5e9 912d4e17 5cdd2a0e 97ddad17
f8441d7e f7310a83 ea9e7dd5 b283f0fb cf194872 b43da5e9 70bc896a 5cdd2a0e
270dce67 f8441d7e 621507ee ea9e7dd5 7564b6c0 cf194872 2f4da1ed 70bc896a
ac12a6c0 270dce67 883afdf0 621507ee 964015e3 7564b6c0 439678ca 2f4da1ed

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 30 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
1bd9e6e3 ac12a6c0 1b9cce4e 883afdf0 0fac4cad 964015e3 b603ab25 439678ca
32418d74 1bd9e6e3 254d8158 1b9cce4e 3f717698 0fac4cad af1cb200 b603ab25
9c89b505 32418d74 b3cdc637 254d8158 38766abf 3f717698 65687d62 af1cb200
3c60352a 9c89b505 831ae864 b3cdc637 8aedd93b 38766abf b4c1fb8b 65687d62
2a116c70 3c60352a 136a0b39 831ae864 476048d4 8aedd93b 55f9c3b3 b4c1fb8b
a0c7c66f 2a116c70 c06a5478 136a0b39 b47a7dc5 476048d4 c9dc576e 55f9c3b3
b7e58f33 a0c7c66f 22d8e054 c06a5478 3a3537a9 b47a7dc5 46a23b02 c9dc576e
79baf4ca b7e58f33 8f8cdf41 22d8e054 9455b731 3a3537a9 ee2da3d3 46a23b02
ad5b0bcf 79baf4ca cb1e676f 8f8cdf41 289d35e0 9455b731 bd49d1a9 ee2da3d3
a167bd76 ad5b0bcf 75e994f3 cb1e676f da27276b 289d35e0 b98ca2ad bd49d1a9

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 40 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2ccc1878 a167bd76 b6179f5a 75e994f3 7eded43b da27276b af0144e9 b98ca2ad
610c6084 2ccc1878 cf7aed42 b6179f5a 9da32cab 7eded43b 3b5ed139 af0144e9
a40209fe 610c6084 9830f059 cf7aed42 7d483846 9da32cab a1dbf6f6 3b5ed139
6fa376a2 a40209fe 18c108c2 9830f059 12a851cf 7d483846 655ced19 a1dbf6f6
53f9ffc5 6fa376a2 0413fd48 18c108c2 c3d3327b 12a851cf c233ea41 655ced19
4f60bbd5 53f9ffc5 46ed44df 0413fd48 f3cae7e6 c3d3327b 8e789542 c233ea41
6e89a7fb 4f60bbd5 f3ff8aa7 46ed44df 17394ca0 f3cae7e6 93de1e99 8e789542
fef3cb16 6e89a7fb c177aa9e f3ff8aa7 4a9e594f 17394ca0 3f379e57 93de1e99
fa8e6731 fef3cb16 134ff6dd c177aa9e 7d9e1966 4a9e594f 6500b9ca 3f379e57
08a826c3 fa8e6731 e7962dfd 134ff6dd ebfa90cc 7d9e1966 ca7a54f2 6500b9ca
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 50 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
614c7627 08a826c3 1cce63f5 e7962dfd 969ecf53 ebfa90cc cb33ecf0 ca7a54f2
d776618d 614c7627 504d8611 1cce63f5 423489f6 969ecf53 86675fd4 cb33ecf0
ef958266 d776618d 98ec4ec2 504d8611 6ef4554d 423489f6 7a9cb4f6 86675fd4
04b44fd2 ef958266 ecc31bae 98ec4ec2 290032b5 6ef4554d 4fb211a4 7a9cb4f6
008d6012 04b44fd2 2b04cddf ecc31bae 50aa1faa 290032b5 aa6b77a2 4fb211a4
57859fec 008d6012 689fa409 2b04cddf c00cd655 50aa1faa 95a94801 aa6b77a2
c864420d 57859fec 1ac02401 689fa409 2fb3c502 c00cd655 fd528550 95a94801
e7423482 c864420d 0b3fd8af 1ac02401 aac3b183 2fb3c502 b2ae0066 fd528550
5c5be9dd e7423482 c8841b90 0b3fd8af 8b1ba117 aac3b183 28117d9e b2ae0066
ebd4948c 5c5be9dd 846905ce c8841b90 74a75fe1 8b1ba117 8c1d561d 28117d9e

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 60 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
05627b53 ebd4948c b7d3bab8 846905ce f58d98d8 74a75fe1 08bc58dd 8c1d561d
28aaec87 05627b53 a92919d7 b7d3bab8 cc6b5f2a f58d98d8 ff0ba53a 08bc58dd
0f92d652 28aaec87 c4f6a60a a92919d7 b8ab6d40 cc6b5f2a c6c7ac6c ff0ba53a
2ad0c8ee 0f92d652 55d90e51 c4f6a60a 69caa1b7 b8ab6d40 f956635a c6c7ac6c
```

A.2.2.2. Message Block 2

A.2.2.2.1. Expanded Message

W_0 W_1 ... W_67:

```
80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000200
80404000 00000000 01008080 10005000 00000000 002002a0 ac545c04 00000000
09582a39 a0003000 00000000 00200280 a4515804 20200040 51609838 30005701
a0002000 008200aa 6ad525d0 0a0e0216 b0f52042 fa7073b0 20000000 008200a8
7a542590 22a20044 d5d6ebd2 82005771 8a202240 b42826aa eaf84e59 4898eaf9
8207283d ee6775fa a3e0e0a0 8828488a 23b45a5d 628a22c4 8d6d0615 38300a7e
e96260e5 2b60c020 502ed531 9e878cb9 218c38f8 dcae3cb7 2a3e0e0a e9e0c461
8c3e3831 44aaa228 dc60a38b 518300f7
```

W'_0 W'_1 ... W'_63:

```
80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000200 80404000 00000000 01008080 10005200
80404000 002002a0 ad54dc84 10005000 09582a39 a02032a0 ac545c04 00200280
ad09723d 80203040 51609838 30205581 04517804 20a200ea 3bb5bde8 3a0e5517
10f50042 faf2731a 4ad525d0 0a8c02be caa105d2 d8d273f4 f5d6ebd2 828257d9
f07407d0 968a26ee 3f2ea58b ca98bd88 08270a7d 5a4f5350 4918aef9 c0b0a273
a1b37260 8ced573e 2e8de6b5 b01842f4 cad63ab8 49eae2e4 dd43d324 a6b786c7
c8ee581d f7cefc97 7a10db3b 776748d8 adb200c9 98049e9f f65ead81 b863c496
```

A.2.2.2.2. Intermediate Values During Iterative Compression

```
   A        B        C        D        E        F        G        H
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 0 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

5950de81 468664eb 42fd4c86 1e7ca00a c0a5910b ae9a55ea 1adb8d17 763ca222
1cc66027 5950de81 0cc9d68d 42fd4c86 24fe81a1 c0a5910b af5574d2 1adb8d17
b7197324 1cc66027 a1bd02b2 0cc9d68d 61b7397a 24fe81a1 885e052c af5574d2
b1aacb3f b7197324 8cc04e39 a1bd02b2 4c7cbb59 61b7397a 0d0927f4 885e052c
920d5d4d b1aacb3f 32e6496e 8cc04e39 c6c863a3 4c7cbb59 cbd30db9 0d0927f4
03162191 920d5d4d 55967f63 32e6496e dbcb73dd c6c863a3 daca63e5 cbd30db9
cbfddbb7 03162191 1aba9b24 55967f63 6a6eaafb dbcb73dd 1d1e3643 daca63e5
67f45147 cbfddbb7 2c432206 1aba9b24 e0cc5b97 6a6eaafb 9eeede5b 1d1e3643
dfc06393 67f45147 fbb76f97 2c432206 9d84a8d5 e0cc5b97 57db5375 9eeede5b
777f980d dfc06393 e8a28ecf fbb76f97 89d0a059 9d84a8d5 dcbf0662 57db5375
502a9be2 777f980d 80c727bf e8a28ecf befc3eda 89d0a059 46acec25 dcbf0662

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 10 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
df0f77ed 502a9be2 ff301aee 80c727bf c8b999f7 befc3eda 02cc4e85 46acec25
b8bc2801 df0f77ed 5537c4a0 ff301aee 3a05da38 c8b999f7 f6d5f7e1 02cc4e85
5b3baaa5 b8bc2801 1eefdbbe 5537c4a0 eebf718f 3a05da38 cfbe45cc f6d5f7e1
0f7185e4 5b3baaa5 78500371 1eefdbbe f3fbf969 eebf718f d1c1d02e cfbe45cc
141cb1e7 0f7185e4 77554ab6 78500371 5cc495db f3fbf969 8c7f75fb d1c1d02e
f185448a 141cb1e7 e30bc81e 77554ab6 32028d02 5cc495db cb4f9fdf 8c7f75fb
a7374acd f185448a 3963ce28 e30bc81e 3d03e81b 32028d02 aedae624 cb4f9fdf
aaca2dcb a7374acd 0a8915e3 3963ce28 130bc932 3d03e81b 68119014 aedae624
3d2dfd31 aaca2dcb 6e959b4e 0a8915e3 07fff8f8 130bc932 40d9e81f 68119014
15bab3e6 3d2dfd31 945b9755 6e959b4e 85b2dd34 07fff8f8 4990985e 40d9e81f

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 20 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
f477625b 15bab3e6 5bfa627a 945b9755 d2b3c82b 85b2dd34 c7c03fff 4990985e
ecbfba29 f477625b 7567cc2b 5bfa627a 604bda38 d2b3c82b e9a42d96 c7c03fff
b9f6943d ecbfba29 eec4b7e8 7567cc2b e996d68b 604bda38 415e959e e9a42d96
c537ac67 b9f6943d 7f7453d9 eec4b7e8 7f6c2bc6 e996d68b d1c3025e 415e959e
c59665b3 c537ac67 ed287b73 7f7453d9 1a89ef0d 7f6c2bc6 b45f4cb6 d1c3025e
50115e1f c59665b3 6f58cf8a ed287b73 3ddf2899 1a89ef0d 5e33fb61 b45f4cb6
44196085 50115e1f 2ccb678b 6f58cf8a 0abc22da 3ddf2899 7868d44f 5e33fb61
bde4e355 44196085 22bc3ea0 2ccb678b da96412a 0abc22da 44c9eef9 7868d44f
ca176dca bde4e355 32c10a88 22bc3ea0 b418ac1b da96412a 16d055e1 44c9eef9
541e456e ca176dca c9c6ab7b 32c10a88 35cf8215 b418ac1b 0956d4b2 16d055e1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 30 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
b6feeef7 541e456e 2edb9594 c9c6ab7b d41f5fda 35cf8215 60dda0c5 0956d4b2
026e42f7 b6feeef7 3c8adca8 2edb9594 c9436b11 d41f5fda 10a9ae7c 60dda0c5
8fd27582 026e42f7 fdddef6d 3c8adca8 a48dc4c2 c9436b11 fed6a0fa 10a9ae7c
2527f8c6 8fd27582 dc85ee04 fdddef6d b29dc9d4 a48dc4c2 588e4a1b fed6a0fa
3218579f 2527f8c6 a4eb051f dc85ee04 0da81ad7 b29dc9d4 2615246e 588e4a1b
35421cf3 3218579f 4ff18c4a a4eb051f 644b37e4 0da81ad7 4ea594ee 2615246e
```

```
12cb048f 35421cf3 30af3e64 4ff18c4a 107cb2fb 644b37e4 d6b86d40 4ea594ee
c6716749 12cb048f 8439e66a 30af3e64 7903974d 107cb2fb bf232259 d6b86d40
66bf4600 c6716749 96091e25 8439e66a e5575380 7903974d 97d883e5 bf232259
046516a9 66bf4600 e2ce938c 96091e25 e23d4f18 e5575380 ba6bc81c 97d883e5

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 40 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
e14ab898 046516a9 7e8c00cd e2ce938c 6e25affe e23d4f18 9c072aba ba6bc81c
bc44d883 e14ab898 ca2d5208 7e8c00cd 4ef0cb38 6e25affe 78c711ea 9c072aba
e017c779 bc44d883 957131c2 ca2d5208 10132c10 4ef0cb38 7ff3712d 78c711ea
11154e38 e017c779 89b10778 957131c2 c1d401bd 10132c10 59c27786 7ff3712d
3ba43e10 11154e38 2f8ef3c0 89b10778 953c1e65 c1d401bd 60808099 59c27786
445e8d34 3ba43e10 2a9c7022 2f8ef3c0 94bcdd11 953c1e65 0dee0ea0 60808099
34d09ee0 445e8d34 487c2077 2a9c7022 1d0ea72c 94bcdd11 f32ca9e0 0dee0ea0
18c77c40 34d09ee0 bd1a6888 487c2077 a8ca98c6 1d0ea72c e88ca5e6 f32ca9e0
a2507cea 18c77c40 a13dc069 bd1a6888 9845362a a8ca98c6 3960e875 e88ca5e6
7e014176 a2507cea 8ef88031 a13dc069 2cb0c2f2 9845362a c6354654 3960e875

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 50 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
eb39074b 7e014176 a0f9d544 8ef88031 0df22b74 2cb0c2f2 b154c229 c6354654
f67597e1 eb39074b 0282ecfc a0f9d544 8d4f6b2f 0df22b74 17916586 b154c229
31e9309d f67597e1 720e97d6 0282ecfc eecf99be 8d4f6b2f 5ba06f91 17916586
c6329c3c 31e9309d eb2fc3ec 720e97d6 c672ad96 eecf99be 597c6a7b 5ba06f91
75cc3800 c6329c3c d2613a63 eb2fc3ec 8515c87f c672ad96 cdf7767c 597c6a7b
925156ad 75cc3800 6538798c d2613a63 150cbd57 8515c87f 6cb63395 cdf7767c
7d0de10b 925156ad 987000eb 6538798c 7ee47610 150cbd57 43fc28ae 6cb63395
2066f136 7d0de10b a2ad5b24 987000eb 7d7aadcc 7ee47610 eab8a865 43fc28ae
85b31359 2066f136 1bc216fa a2ad5b24 07b9cfd1 7d7aadcc b083f723 eab8a865
6cddcb93 85b31359 cde26c40 1bc216fa c43eb29c 07b9cfd1 6e63ebd5 b083f723

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 60 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
23eff97d 6cddcb93 6626b30b cde26c40 1ea21d46 c43eb29c 7e883dce 6e63ebd5
07bd4e82 23eff97d bb9726d9 6626b30b c8d6867c 1ea21d46 94e621f5 7e883dce
64f3dc4a 07bd4e82 dff2fa47 bb9726d9 96e4028f c8d6867c ea30f510 94e621f5
87ee4178 64f3dc4a 7a9d040f dff2fa47 af7ee1ee 96e4028f 33e646b4 ea30f510
```

A.2.3. Hash Value

```
debe9ff9 2275b8a1 38604889 c18e5a4d 6fdb70e5 387e5765 293dcba3 9c0c5732
```

Appendix B. Example Results

These examples only provide results of hashing, and can be found in the Botan [BOTAN], OpenSSL [OPENSSL] and GmSSL [GMSSL] cryptographic libraries.

B.1. GB/T 32918.2-2016 A.2 Example 1

From [GBT.32918.2-2016] A.2, "Z_A = H_256(ENTL_A || ID_A || a || b || x_G || y_G || x_A || y_A)".

Input:

```
0090 414C494345313233405941484F4F2E434F4D
787968B4 FA32C3FD 2417842E 73BBFEFF 2F3C848B 6831D7E0 EC65228B 3937E498
63E4C6D3 B23B0C84 9CF84241 484BFE48 F61D59A5 B16BA06E 6E12D1DA 27C5249A
421DEBD6 1B62EAB6 746434EB C3CC315E 32220B3B ADD50BDC 4C4E6C14 7FEDD43D
0680512B CBB42C07 D47349D2 153B70C4 E5D7FDFC BFA36EA1 A85841B9 E46E09A2
0AE4C779 8AA0F119 471BEE11 825BE462 02BB79E2 A5844495 E97C04FF 4DF2548A
7C0240F8 8F1CD4E1 6352A73C 17B7F16F 07353E53 A176D684 A9FE0C6B B798E857
```

Output:

```
F4A38489 E32B45B6 F876E3AC 2168CA39 2362DC8F 23459C1D 1146FC3D BFB7BC9A
```

B.2. GB/T 32918.2-2016 A.2 Example 2

From [GBT.32918.2-2016] A.2, "e = H_256(M)".

Input:

```
F4A38489 E32B45B6 F876E3AC 2168CA39 2362DC8F 23459C1D 1146FC3D BFB7BC9A
6D657373 61676520 64696765 7374
```

Output:

```
B524F552 CD82B8B0 28476E00 5C377FB1 9A87E6FC 682D48BB 5D42E3D9 B9EFFE76
```

B.3. GB/T 32918.2-2016 A.3 Example 1

From [GBT.32918.2-2016] A.3, "Z_A = H_256(ENTL_A || ID_A || a || b || x_G || y_G || x_A || y_A)".

Input:

```
0090 414C494345313233405941484F4F2E434F4D
00
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00
E78BCD09 746C2023 78A7E72B 12BCE002 66B9627E CB0B5A25 367AD1AD 4CC6242B
00
CDB9CA7F 1E6B0441 F658343F 4B10297C 0EF9B649 1082400A 62E7A748 5735FADD
01
3DE74DA6 5951C4D7 6DC89220 D5F7777A 611B1C38 BAE260B1 75951DC8 060C2B3E
01
65961645 281A8626 607B917F 657D7E93 82F1EA5C D931F40F 6627F357 542653B2
01
68652213 0D590FB8 DE635D8F CA715CC6 BF3D05BE F3F75DA5 D5434544 48166612
```

Output:

```
26352AF8 2EC19F20 7BBC6F94 74E11E90 CE0F7DDA CE03B27F 801817E8 97A81FD5
```

B.4. GB/T 32918.2-2016 A.3 Example 2

From [GBT.32918.2-2016] A.3, "e = H_256(M)".

Input:

```
26352AF8 2EC19F20 7BBC6F94 74E11E90 CE0F7DDA CE03B27F 801817E8 97A81FD5
6D657373 61676520 64696765 7374
```

Output:

```
AD673CBD A3114171 29A9EAA5 F9AB1AA1 633AD477 18A84DFD 46C17C6F A0AA3B12
```

B.5. GB/T 32918.3-2016 A.2 Example 1

From [GBT.32918.3-2016] A.2, "Z_A = H_256(ENTL_A || ID_A || a || b || x_G || y_G || x_A || y_A)".

Input:

```
0090 414C 49434531 32334059 41484F4F 2E434F4D
787968B4 FA32C3FD 2417842E 73BBFEFF 2F3C848B 6831D7E0 EC65228B 3937E498
63E4C6D3 B23B0C84 9CF84241 484BFE48 F61D59A5 B16BA06E 6E12D1DA 27C5249A
421DEBD6 1B62EAB6 746434EB C3CC315E 32220B3B ADD50BDC 4C4E6C14 7FEDD43D
0680512B CBB42C07 D47349D2 153B70C4 E5D7FDFC BFA36EA1 A85841B9 E46E09A2
3099093B F3C137D8 FCBBCDF4 A2AE50F3 B0F216C3 122D7942 5FE03A45 DBFE1655
3DF79E8D AC1CF0EC BAA2F2B4 9D51A4B3 87F2EFAF 48233908 6A27A8E0 5BAED98B
```

Output:

```
E4D1D0C3 CA4C7F11 BC8FF8CB 3F4C02A7 8F108FA0 98E51A66 8487240F 75E20F31
```

B.6. GB/T 32918.3-2016 A.2 Example 2

From [GBT.32918.3-2016] A.2, "Z_B = H_256(ENTL_B || ID_B || a || b || x_G || y_G || x_B || y_B)".

Input:

```
0088 42 494C4C34 35364059 41484F4F 2E434F4D
787968B4 FA32C3FD 2417842E 73BBFEFF 2F3C848B 6831D7E0 EC65228B 3937E498
63E4C6D3 B23B0C84 9CF84241 484BFE48 F61D59A5 B16BA06E 6E12D1DA 27C5249A
421DEBD6 1B62EAB6 746434EB C3CC315E 32220B3B ADD50BDC 4C4E6C14 7FEDD43D
0680512B CBB42C07 D47349D2 153B70C4 E5D7FDFC BFA36EA1 A85841B9 E46E09A2
245493D4 46C38D8C C0F11837 4690E7DF 633A8A4B FB3329B5 ECE604B2 B4F37F43
53C0869F 4B9E1777 3DE68FEC 45E14904 E0DEA45B F6CECF99 18C85EA0 47C60A4C
```

Output:

```
6B4B6D0E 276691BD 4A11BF72 F4FB501A E309FDAC B72FA6CC 336E6656 119ABD67
```

B.7. GB/T 32918.3-2016 A.2 Example 3

From [GBT.32918.3-2016] A.2, "Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2)".

Input:

```
47C82653 4DC2F6F1 FBF28728 DD658F21 E174F481 79ACEF29 00F8B7F5 66E40905
E4D1D0C3 CA4C7F11 BC8FF8CB 3F4C02A7 8F108FA0 98E51A66 8487240F 75E20F31
6B4B6D0E 276691BD 4A11BF72 F4FB501A E309FDAC B72FA6CC 336E6656 119ABD67
6CB56338 16F4DD56 0B1DEC45 8310CBCC 6856C095 05324A6D 23150C40 8F162BF0
0D6FCF62 F1036C0A 1B6DACCF 57399223 A65F7D7B F2D9637E 5BBBEB85 7961BF1A
1799B2A2 C7782953 00D9A232 5C686129 B8F2B533 7B3DCF45 14E8BBC1 9D900EE5
54C9288C 82733EFD F7808AE7 F27D0E73 2F7C73A7 D9AC98B7 D8740A91 D0DB3CF4
```

Output:

```
FF49D95B D45FCE99 ED54A8AD 7A709110 9F513944 42916BD1 54D1DE43 79D97647
```

B.8. GB/T 32918.3-2016 A.2 Example 4

From [GBT.32918.3-2016] A.2, "S_B = 0x02 || y_V || Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2)".

Input:

```
02
2AF86EFE 732CF12A D0E09A1F 2556CC65 0D9CCCE3 E249866B BB5C6846 A4C4A295
FF49D95B D45FCE99 ED54A8AD 7A709110 9F513944 42916BD1 54D1DE43 79D97647
```

Output:

```
284C8F19 8F141B50 2E81250F 1581C7E9 EEB4CA69 90F9E02D F388B454 71F5BC5C
```

B.9. GB/T 32918.3-2016 A.2 Example 5

From [GBT.32918.3-2016] A.2, "S_A = 0x03 || y_V || Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2)".

Input:

```
03
2AF86EFE 732CF12A D0E09A1F 2556CC65 0D9CCCE3 E249866B BB5C6846 A4C4A295
FF49D95B D45FCE99 ED54A8AD 7A709110 9F513944 42916BD1 54D1DE43 79D97647
```

Output:

```
23444DAF 8ED75343 66CB901C 84B3BDBB 63504F40 65C1116C 91A4C006 97E6CF7A
```

B.10. GB/T 32918.3-2016 A.3 Example 2

From [GBT.32918.3-2016] A.3, "Z_B = H_256(ENTL_B || ID_B || a || b || x_G || y_G || x_B || y_B)".

Input:

```
0088 42 494C4C34 35364059 41484F4F 2E434F4D
00
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00
E78BCD09 746C2023 78A7E72B 12BCE002 66B9627E CB0B5A25 367AD1AD 4CC6242B
00
CDB9CA7F 1E6B0441 F658343F 4B10297C 0EF9B649 1082400A 62E7A748 5735FADD
01
3DE74DA6 5951C4D7 6DC89220 D5F7777A 611B1C38 BAE260B1 75951DC8 060C2B3E
00
34297DD8 3AB14D5B 393B6712 F32B2F2E 938D4690 B095424B 89DA880C 52D4A7D9
01
99BBF11A C95A0EA3 4BBD00CA 50B93EC2 4ACB6833 5D20BA5D CFE3B33B DBD2B62D
```

Output:

```
557BAD30 E183559A EEC3B225 6E1C7C11 F870D22B 165D015A CF9465B0 9B87B527
```

B.11. GB/T 32918.3-2016 A.3 Example 3

From [GBT.32918.3-2016] A.3, "Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2)".

Input:

```
00DADD08 7406221D 657BC3FA 79FF329B B022E9CB 7DDFCFCC FE277BE8 CD4AE9B9
54ECF008 0215977B 2E5D6D61 B98A9944 2F03E880 3DC39E34 9F8DCA56 21A9ACDF
2B557BAD 30E18355 9AEEC3B2 256E1C7C 11F870D2 2B165D01 5ACF9465 B09B87B5
27018107 6543ED19 058C38B3 13D73992 1D46B800 94D961A1 3673D4A5 CF8C7159
E30401D8 CFFF7CA2 7A01A2E8 8C186737 48FDE9A7 4C1F9B45 646ECA09 97293C15
C34DD800 2A4832B4 DCD399BA AB3FFFE7 DD6CE6ED 68CC43FF A5F2623B 9BD04E46
8D322A2A 0016599B B52ED9EA FAD01CFA 453CF305 2ED60184 D2EECFD4 2B52DB74
110B984C 23
```

Output:

```
E05FE287 B73B0CE6 639524CD 86694311 562914F4 F6A34241 01D885F8 8B05369C
```

B.12. GB/T 32918.3-2016 A.3 Example 4

From [GBT.32918.3-2016] A.3, "S_B = 0x02 || y_V || Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2)".

Input:

```
02
01
F0464B1E 81684E5E D6EF281B 55624EF4 6CAA3B2D 37484372 D91610B6 98252CC9
E05FE287 B73B0CE6 639524CD 86694311 562914F4 F6A34241 01D885F8 8B05369C
```

Output:

```
4EB47D28 AD3906D6 244D01E0 F6AEC73B 0B51DE15 74C13798 184E4833 DBAE295A
```

B.13. GB/T 32918.3-2016 A.3 Example 5

From [GBT.32918.3-2016] A.3, "S_A = 0x03 || y_V || Hash(x_V || Z_A || Z_B || x_1 || y_1 || x_2 || y_2)".

Input:

```
03
01
F0464B1E 81684E5E D6EF281B 55624EF4 6CAA3B2D 37484372 D91610B6 98252CC9
E05FE287 B73B0CE6 639524CD 86694311 562914F4 F6A34241 01D885F8 8B05369C
```

Output:

588AA670 64F24DC2 7CCAA1FA B7E27DFF 811D500A D7EF2FB8 F69DDF48 CC0FECB7

B.14. GB/T 32918.4-2016 A.2 Example 1

From [GBT.32918.4-2016], "C_3 = Hash(x_2 || M || y_2)".

Input:

57E7B636 23FAE5F0 8CDA468E 872A20AF A03DED41 BF140377 656E6372 79707469
6F6E2073 74616E64 6172640E 040DC83A F31A6799 1F2B01EB F9EFD888 1F0A0493
000603

Output:

6AFB3BCE BD76F82B 252CE5EB 25B57996 86902B8C F2FD8753 6E55EF76 03B09E7C

B.15. GB/T 32918.4-2016 A.2 Example 2

From [GBT.32918.4-2016], "C_3 = Hash(x_2 || M || y_2)".

Input:

64D20D27 D0632957 F8028C1E 024F6B02 EDF23102 A566C932 AE8BD613 A8E865FE
656E6372 79707469 6F6E2073 74616E64 61726458 D225ECA7 84AE300A 81A2D482
81A828E1 CEDF11C4 21909984 02653750 77BF78

Output:

9C3D7360 C30156FA B7C80A02 76712DA9 D8094A63 4B766D3A 285E0748 0653426D

B.16. GB/T 32918.4-2016 A.3 Example 1

From [GBT.32918.4-2016], "C_3 = Hash(x_2 || M || y_2)".

Input:

01C6271B 31F6BE39 6A4166C0 616CF4A8 ACDA5BEF 4DCBF2DD 42656E63 72797074
696F6E20 7374616E 64617264 0147AF35 DFA1BFE2 F161521B CF59BAB8 3564868D
92958817 35

Output:

F0A41F6F 48AC723C ECFC4B76 7299A5E2 5C064167 9FBD2D4D 20E9FFD5 B9F0DAB8

B.17. GB/T 32918.4-2016 A.3 Example 2

From [GBT.32918.4-2016], "C_3 = Hash(x_2 || M || y_2)".

Input:

```
0083E628 CF701EE3 141E8873 FE55936A DF24963F 5DC9C648 0566C80F 8A1D8CC5
1B656E63 72797074 696F6E20 7374616E 64617264 01524C64 7F0C0412 DEFD468B
DA3AE0E5 A80FCC8F 5C990FEE 11602929 232DCD9F 36
```

Output:

```
73A48625 D3758FA3 7B3EAB80 E9CFCABA 665E3199 EA15A1FA 8189D96F 579125E4
```

Appendix C. Sample Implementation In C

This sample implementation is used to generate the examples given in this document.

C.1. sm3.h

"sm3.h" is the header file for the SM3 function.

```
<CODE BEGINS>
#ifndef SM3_H
#define SM3_H
#include <stdio.h>
#include <inttypes.h>

#define SM3_BLOCK_SIZE_IN_BYTES 64
#define SM3_BLOCK_SIZE_IN_32 16

typedef struct {
  uint32_t state[8];
  uint64_t bitcount;
  uint32_t buffer[16];
} sm3_context;

void sm3(
    unsigned char *message,
    int message_length,
    unsigned char *digest /* 256-bits */);

#endif
<CODE ENDS>
```

C.2. sm3.c

"sm3.c" contains the main implementation of SM3.

```
<CODE BEGINS>
/* A sample implementation of SM3 */

#include <stdlib.h>
#include <string.h>
#include "sm3.h"
#include "print.h"

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

/* Functions for SM3 algorithm */
#define FF1(X,Y,Z) ((X) ^ (Y) ^ (Z))
#define FF2(X,Y,Z) (((X) & (Y)) | ((X) & (Z)) | ((Y) & (Z)))
#define GG1(X,Y,Z) ((X) ^ (Y) ^ (Z))
#define GG2(X,Y,Z) (((X) & (Y)) | ((~X) & (Z)))
#define P0(X) ((X) ^ ROTL32((X),  9) ^ ROTL32((X), 17))
#define P1(X) ((X) ^ ROTL32((X), 15) ^ ROTL32((X), 23))

typedef union sm3_block {
  uint32_t content[16];
  struct {
    uint32_t content[14];
    uint64_t length;
  } last_block;
} sm3_block_t;

typedef struct sm3_padded_blocks {
  sm3_block_t** blocks;
  int bitcount;
  int n;
  sm3_block_t last_blocks[2];
} sm3_pb_t;

/* Initialize the context */
static void sm3_init(sm3_context *ctx)
{
  const uint32_t IV[8] = {
    0x7380166f, 0x4914b2b9, 0x172442d7, 0xda8a0600,
    0xa96f30bc, 0x163138aa, 0xe38dee4d, 0xb0fb0e4e
  };

  int i;
```

```
  for (i = 0; i < 8; i++)
  {
    ctx->state[i] = IV[i];
  }

  debug_print("IV:\n");
  print_hash((unsigned*)ctx->state);

  memset(ctx->buffer, 0, sizeof(uint32_t) * 16);

  ctx->bitcount = 0;
}

static void sm3_me(sm3_context *ctx, uint32_t W[], uint32_t WP[])
{
  int i;

  debug_print("\n=== SM3 Message Expansion ME (sm3_me):\n");

  /* Message Expansion ME */
  for (i = 0; i < 16; i++)
  {
    W[i] = ctx->buffer[i];
  }

  for (i = 16; i < 68; i++)
  {
    W[i] = P1(W[i - 16] ^ W[i - 9] ^ ROTL32(W[i - 3], 15)) ^
           ROTL32(W[i - 13], 7) ^ W[i - 6];
  }

  for (i = 0; i < 64; i++)
  {
    WP[i] = W[i] ^ W[i + 4];
  }

  debug_print("\nME(m'): W_0 W_1 ... W_67:\n");
  print_block((unsigned*)W, 68);

  debug_print("\nME(m'): W'_0 W'_1 ... W'_63:\n");
  print_block((unsigned*)WP, 64);
}

static void sm3_cf(sm3_context *ctx, uint32_t W[], uint32_t WP[])
{
  uint32_t A, B, C, D, E, F, G, H;
  uint32_t SS1, SS2, TT1, TT2, Tj;
```

```
   int i;

   debug_print("\n=== SM3 Compression Function CF (sm3_cf):\n");

   A = ctx->state[0];
   B = ctx->state[1];
   C = ctx->state[2];
   D = ctx->state[3];
   E = ctx->state[4];
   F = ctx->state[5];
   G = ctx->state[6];
   H = ctx->state[7];

   debug_print("   A        B        C        D        "
       "E        F        G        H\n");
   debug_print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
       " initial value "
       "~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n");
   print_af(i, A, B, C, D, E, F, G, H);

   /* Compression Function */
   for (i = 0; i < 64; i++)
   {
     if (i < 16)
     {
       Tj = 0x79cc4519;
       SS1 = ROTL32(ROTL32(A, 12) + E + ROTL32(Tj, i), 7);
       SS2 = SS1 ^ ROTL32(A, 12);
       TT1 = FF1(A, B, C) + D + SS2 + WP[i];
       TT2 = GG1(E, F, G) + H + SS1 + W[i];
     }
     else
     {
       Tj = 0x7a879d8a;
       SS1 = ROTL32(ROTL32(A, 12) + E + ROTL32(Tj, i), 7);
       SS2 = SS1 ^ ROTL32(A, 12);
       TT1 = FF2(A, B, C) + D + SS2 + WP[i];
       TT2 = GG2(E, F, G) + H + SS1 + W[i];
     }

     D = C;
     C = ROTL32(B, 9);
     B = A;
     A = TT1;
     H = G;
     G = ROTL32(F, 19);
     F = E;
     E = P0(TT2);
```

```
    print_af(i, A, B, C, D, E, F, G, H);
  }

  /* Update Context */
  ctx->state[0] ^= A;
  ctx->state[1] ^= B;
  ctx->state[2] ^= C;
  ctx->state[3] ^= D;
  ctx->state[4] ^= E;
  ctx->state[5] ^= F;
  ctx->state[6] ^= G;
  ctx->state[7] ^= H;

}

/*
 * Processes a single 512b block and updates context state
 */
static void sm3_block(sm3_context *ctx)
{
  uint32_t W[68] = {0},
           WP[64] = {0};

  debug_print("\n== ----- SM3 Process Block (sm3_block) begin ----\n");
  debug_print("Context Initial State:\n");
  print_block((unsigned*)ctx->state, 8);

  debug_print("Block Input:\n");
  print_block((unsigned*)ctx->buffer, 16);

  sm3_me(ctx, W, WP);
  sm3_cf(ctx, W, WP);

  debug_print("\n~~~~~~~~~~~~~~~~~~~"
      " final block hash value (V_64) "
      "~~~~~~~~~~~~~~~~~~~\n");
  print_block((unsigned*)ctx->state, 8);
  debug_print("== ----- SM3 Process Block (sm3_block) end -----\n\n");
}

uint32_t sm3_end_bytes(uint32_t *input, int length)
{
  uint32_t output = 0;
  uint8_t *b;

  // Apply the "1" right after the message
  b = (uint8_t*)&output;
  switch (length) {
```

```
    case 0:
      b[3] = (0x80);
      break;

    case 1:
      b[3] = (uint8_t)*input;
      b[2] = (0x80);
      break;

    case 2:
      b[3] = *(uint8_t*)input;
      b[2] = *((uint8_t*)input + 1);
      b[1] = (0x80);
      break;

    case 3:
      b[3] = *(uint8_t*)input;
      b[2] = *((uint8_t*)input + 1);
      b[1] = *((uint8_t*)input + 2);
      b[0] = (0x80);
      break;
  }

  /*
  debug_print("\n~~~~~~~~~~~~~~~~~~"
      " sm3_end_bytes input(len(%i), %0.8x), output(%0.8x)"
      "~~~~~~~~~~~~~~~~~~~~\n", length, *input, output);
   */

  return output;
}


/*
 * Splits a message into blocks and adds padding into blocks of
 * 512 bits. `length` is in bytes (64 => 512 bits).
 */
static int *sm3_pad_blocks(sm3_pb_t* result,
  unsigned char *message,
  int length,
  sm3_context *ctx)
{
  uint32_t *read_p = 0;
  uint32_t *write_p = 0;
  int i, j, n, remaining_bytes;

  debug_print("\n=== SM3 Pad Input Into Blocks (sm3_pad_blocks):\n");
```

```
   /* number of blocks */
   /* 512, 512, last block is max 446 */
   remaining_bytes = length;
   n = length / 64;

   read_p = (uint32_t*)message;
   write_p = (uint32_t*)(result->last_blocks[0].content);

   debug_print("\n==== Full Blocks (%i)\n", n);

   /* process and gather full 512-bit blocks */
   for (i = 0; i < n; i++)
   {
     result->blocks[i] = (sm3_block_t*)read_p;
     read_p += SM3_BLOCK_SIZE_IN_32;
     remaining_bytes -= SM3_BLOCK_SIZE_IN_BYTES;
     result->bitcount += SM3_BLOCK_SIZE_IN_BYTES * 8;
     result->n = i+1;
   }

   /*
    * Process last block.
    *
    * The last block must contain between 0 to 446 bits of content.
    * If there are between 446 to 512 bits of content ("overflow"), we
    * need an extra block.
    */

   /*
    * Copy out the last blocks so we can pad them in a new buffer.
    */
   for (j = 0; j < remaining_bytes / 4 /* u32 */; j++)
   {
     result->last_blocks[0].content[j] = read_p[j];
   }

   /* write "10" bit */
   read_p = &read_p[j];
   result->last_blocks[0].content[j] =
     sm3_end_bytes(read_p, remaining_bytes % 4);

   result->blocks[n] = &(result->last_blocks[0]);
   result->n = ++n;
   result->bitcount += remaining_bytes * 8;


   /*
    * This block has no overflow, just write length and return.
```

```
    */
   if (remaining_bytes < 56)
   {
     debug_print("==== Padded Block (%i), last block (%i-bytes)\n",
         1, remaining_bytes);

     /* write length in bits */
     result->last_blocks[0].last_block.length =
       (uint64_t)(length * 8) << 32;

     return 0;
   }

   /*
    * This last block has overflowed.
    * (i.e., it contains 446 to 512 bits of content)
    *
    * We pad the last packet with the x80, then 0's, and move the length
    * to the next packet.
    */
   debug_print("==== Padded Blocks (%i), with "
       "overflow block (%i-bytes)\n",
       2, remaining_bytes);

   result->blocks[n] = &(result->last_blocks[1]);
   result->n = ++n;

   /* write length in bits */
   result->last_blocks[1].last_block.length =
     (uint64_t)(length * 8) << 32;


   return 0;
 }

 /*
  * The SM3 256 Hash Function
  * message: pointer to input message
  * length: length of input message, in bytes
  * digest: final hash of 256-bits, must be 16-bytes long
  */
 void sm3(
   unsigned char *message,
   int length,
   unsigned char *digest /* 256-bits */
 )
 {
   sm3_context ctx;
```

```
   sm3_pb_t result = {0}; /* array of blocks to return */
   int i = 0, j = 0, block = 0;

   if (length == 0)
   {
     return;
   }

   debug_print("= Stage 0: Initialize Context.");
   sm3_init(&ctx);

   debug_print("= Stage 1: Pad Message...\n");

   /* number of full blocks */
   result.blocks = calloc((length + 2), sizeof(uint32_t*));

   sm3_pad_blocks(&result, message, length, &ctx);
   ctx.bitcount = result.bitcount;

   debug_print("==> Split/result into (N=%i) blocks.\n", result.n);
   for (i = 0; i < result.n; i++)
   {
     debug_print("\n== -------- PADDED BLOCK %i of %i --------\n",
         i+1, result.n);
     print_bytes((unsigned*)(result.blocks[i]), 64);
     debug_print("== -------- END PADDED BLOCK %i of %i --------\n",
         i+1, result.n);
   }

   debug_print("= Stage 2: Processing blocks.\n");
   for (i = 0; i < result.n; i++)
   {
     /* Load block into memory */
     for (j = 0; j < 16; j++)
     {
       ctx.buffer[j] = (uint32_t)(result.blocks[i]->content[j]);
     }

     /* Process loaded block */
     block++;
     debug_print("== Processing block %i of N(%i) blocks.\n",
       i, result.n);
     sm3_block(&ctx);
   }

   free(result.blocks);

   debug_print("== Stage 2: Processing blocks done.");
```

```
  debug_print("\n++++++++++++++++++++++"
      " hash value of all blocks "
      "++++++++++++++++++++++\n");
  print_hash((unsigned*)ctx.state);

  memcpy(digest, ctx.state, sizeof(uint32_t) * 8);
}
<CODE ENDS>
```

C.3. sm3_main.c

"sm3_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 "sm3.h"
#include "print.h"

typedef struct {
  uint32_t *message;
  uint32_t *expected;
  int       length;
} test_case;

int sm3_run_example(test_case tc)
{
  unsigned char digest[32] = {0};

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

  sm3(
    (unsigned char*)tc.message,
    tc.length,
    (unsigned char*)digest
  );

  debug_print("++++++++++++++++++++++++++++++"
      " RESULT "
      "++++++++++++++++++++++++++++++\n");
  debug_print("RESULTS:\n");
```

```
    debug_print(" Expected:\n");
    print_hash((unsigned*)tc.expected);

    debug_print(" Digest:\n");
    print_hash((unsigned*)digest);

    return memcmp((unsigned char*)digest,
                  (unsigned char*)tc.expected, 32);
  }

  int main(int argc, char **argv)
  {
    /*
     * This test vector comes from Example 1 of GB/T 3290X-2016,
     * and described in Internet Draft draft-oscca-cfrg-sm3-XX.
     */

    int i;
    test_case tests[20] = {0};

    /* Example 1, From GB/T 32905-2016 */
    /* "abc" */
    static const uint32_t gbt32905m01[1] = { 0x00636261 };
    static const uint32_t gbt32905e01[8] = {
      0x66c7f0f4, 0x62eeedd9, 0xd1f2d46b, 0xdc10e4e2,
      0x4167c487, 0x5cf2f7a2, 0x297da02b, 0x8f4ba8e0
    };
    static const int gbt32905l01 = 3;
    test_case gbt32905t01 = {
      (uint32_t*)&gbt32905m01,
      (uint32_t*)&gbt32905e01,
      gbt32905l01
    };
    tests[0] = gbt32905t01;

    /* Example 2, From GB/T 32905-2016 */
    /*
     * "abcdabcdabcdabcdabcdabcdabcdabcd" +
     * "abcdabcdabcdabcdabcdabcdabcdabcd"
     */
    static const uint32_t gbt32905m02[16] = {
      0x61626364, 0x61626364, 0x61626364, 0x61626364,
      0x61626364, 0x61626364, 0x61626364, 0x61626364,
      0x61626364, 0x61626364, 0x61626364, 0x61626364,
      0x61626364, 0x61626364, 0x61626364, 0x61626364
    };
    static const uint32_t gbt32905e02[8] = {
      0xdebe9ff9, 0x2275b8a1, 0x38604889, 0xc18e5a4d,
```

```
  0x6fdb70e5, 0x387e5765, 0x293dcba3, 0x9c0c5732
};
static const int gbt32905l02 = 64;
test_case gbt32905t02 = {
  (uint32_t*)&gbt32905m02,
  (uint32_t*)&gbt32905e02,
  gbt32905l02
};
tests[1] = gbt32905t02;

/*
 * GB/T 32918.2-2016 A.2 Example 1
 */
static const uint32_t gbt329182m01[53] = {
  0x0090414C, 0x49434531, 0x32334059, 0x41484F4F,
  0x2E434F4D, 0x787968B4, 0xFA32C3FD, 0x2417842E,
  0x73BBFEFF, 0x2F3C848B, 0x6831D7E0, 0xEC65228B,
  0x3937E498, 0x63E4C6D3, 0xB23B0C84, 0x9CF84241,
  0x484BFE48, 0xF61D59A5, 0xB16BA06E, 0x6E12D1DA,
  0x27C5249A, 0x421DEBD6, 0x1B62EAB6, 0x746434EB,
  0xC3CC315E, 0x32220B3B, 0xADD50BDC, 0x4C4E6C14,
  0x7FEDD43D, 0x0680512B, 0xCBB42C07, 0xD47349D2,
  0x153B70C4, 0xE5D7FDFC, 0xBFA36EA1, 0xA85841B9,
  0xE46E09A2, 0x0AE4C779, 0x8AA0F119, 0x471BEE11,
  0x825BE462, 0x02BB79E2, 0xA5844495, 0xE97C04FF,
  0x4DF2548A, 0x7C0240F8, 0x8F1CD4E1, 0x6352A73C,
  0x17B7F16F, 0x07353E53, 0xA176D684, 0xA9FE0C6B,
  0xB798E857
};
static const uint32_t gbt329182e01[8] = {
  0xF4A38489, 0xE32B45B6, 0xF876E3AC, 0x2168CA39,
  0x2362DC8F, 0x23459C1D, 0x1146FC3D, 0xBFB7BC9A
};
static const int gbt329182l01 = 212;
test_case gbt329182t01 = {
  (uint32_t*)&gbt329182m01,
  (uint32_t*)&gbt329182e01,
  gbt329182l01
};
tests[2] = gbt329182t01;

/*
 * GB/T 32918.2-2016 A.2 Example 2
 */
static const uint32_t gbt329182m02[12] = {
  0xF4A38489, 0xE32B45B6, 0xF876E3AC, 0x2168CA39,
  0x2362DC8F, 0x23459C1D, 0x1146FC3D, 0xBFB7BC9A,
  0x6D657373, 0x61676520, 0x64696765, 0x00007473
```

```
};
static const uint32_t gbt329182e02[8] = {
  0xB524F552, 0xCD82B8B0, 0x28476E00, 0x5C377FB1,
  0x9A87E6FC, 0x682D48BB, 0x5D42E3D9, 0xB9EFFE76
};
static const int gbt329182l02 = 46;
test_case gbt329182t02 = {
  (uint32_t*)&gbt329182m02,
  (uint32_t*)&gbt329182e02,
  gbt329182l02
};
tests[3] = gbt329182t02;

/*
 * GB/T 32918.2-2016 A.3 Example 1
 */
static const uint32_t gbt329182m03[55] = {
  0x0090414C, 0x49434531, 0x32334059, 0x41484F4F,
  0x2E434F4D, 0x00000000, 0x00000000, 0x00000000,
  0x00000000, 0x00000000, 0x00000000, 0x00000000,
  0x00000000, 0x0000E78B, 0xCD09746C, 0x202378A7,
  0xE72B12BC, 0xE00266B9, 0x627ECB0B, 0x5A25367A,
  0xD1AD4CC6, 0x242B00CD, 0xB9CA7F1E, 0x6B0441F6,
  0x58343F4B, 0x10297C0E, 0xF9B64910, 0x82400A62,
  0xE7A74857, 0x35FADD01, 0x3DE74DA6, 0x5951C4D7,
  0x6DC89220, 0xD5F7777A, 0x611B1C38, 0xBAE260B1,
  0x75951DC8, 0x060C2B3E, 0x01659616, 0x45281A86,
  0x26607B91, 0x7F657D7E, 0x9382F1EA, 0x5CD931F4,
  0x0F6627F3, 0x57542653, 0xB2016865, 0x22130D59,
  0x0FB8DE63, 0x5D8FCA71, 0x5CC6BF3D, 0x05BEF3F7,
  0x5DA5D543, 0x45444816, 0x00001266
};
static const uint32_t gbt329182e03[8] = {
  0x26352AF8, 0x2EC19F20, 0x7BBC6F94, 0x74E11E90,
  0xCE0F7DDA, 0xCE03B27F, 0x801817E8, 0x97A81FD5
};
static const int gbt329182l03 = 218;
test_case gbt329182t03 = {
  (uint32_t*)&gbt329182m03,
  (uint32_t*)&gbt329182e03,
  gbt329182l03
};
tests[4] = gbt329182t03;

/*
 * GB/T 32918.2-2016 A.3 Example 2
 */
static const uint32_t gbt329182m04[12] = {
```

```
  0x26352AF8, 0x2EC19F20, 0x7BBC6F94, 0x74E11E90,
  0xCE0F7DDA, 0xCE03B27F, 0x801817E8, 0x97A81FD5,
  0x6D657373, 0x61676520, 0x64696765, 0x00007473
};
static const uint32_t gbt329182e04[8] = {
  0xAD673CBD, 0xA3114171, 0x29A9EAA5, 0xF9AB1AA1,
  0x633AD477, 0x18A84DFD, 0x46C17C6F, 0xA0AA3B12
};
static const int gbt329182l04 = 46;
test_case gbt329182t04 = {
  (uint32_t*)&gbt329182m04,
  (uint32_t*)&gbt329182e04,
  gbt329182l04
};
tests[5] = gbt329182t04;

/*
 * GB/T 32918.3-2016 A.2 Example 1
 */
static const uint32_t gbt329183m01[53] = {
  0x0090414C, 0x49434531, 0x32334059, 0x41484F4F,
  0x2E434F4D, 0x787968B4, 0xFA32C3FD, 0x2417842E,
  0x73BBFEFF, 0x2F3C848B, 0x6831D7E0, 0xEC65228B,
  0x3937E498, 0x63E4C6D3, 0xB23B0C84, 0x9CF84241,
  0x484BFE48, 0xF61D59A5, 0xB16BA06E, 0x6E12D1DA,
  0x27C5249A, 0x421DEBD6, 0x1B62EAB6, 0x746434EB,
  0xC3CC315E, 0x32220B3B, 0xADD50BDC, 0x4C4E6C14,
  0x7FEDD43D, 0x0680512B, 0xCBB42C07, 0xD47349D2,
  0x153B70C4, 0xE5D7FDFC, 0xBFA36EA1, 0xA85841B9,
  0xE46E09A2, 0x3099093B, 0xF3C137D8, 0xFCBBCDF4,
  0xA2AE50F3, 0xB0F216C3, 0x122D7942, 0x5FE03A45,
  0xDBFE1655, 0x3DF79E8D, 0xAC1CF0EC, 0xBAA2F2B4,
  0x9D51A4B3, 0x87F2EFAF, 0x48233908, 0x6A27A8E0,
  0x5BAED98B
};
static const uint32_t gbt329183e01[8] = {
  0xE4D1D0C3, 0xCA4C7F11, 0xBC8FF8CB, 0x3F4C02A7,
  0x8F108FA0, 0x98E51A66, 0x8487240F, 0x75E20F31
};
static const int gbt329183l01 = 212;
test_case gbt329183t01 = {
  (uint32_t*)&gbt329183m01,
  (uint32_t*)&gbt329183e01,
  gbt329183l01
};
tests[6] = gbt329183t01;

/*
```

```
 * GB/T 32918.3-2016 A.2 Example 2
 */
static const uint32_t gbt329183m02[53] = {
  0x00884249, 0x4C4C3435, 0x36405941, 0x484F4F2E,
  0x434F4D78, 0x7968B4FA, 0x32C3FD24, 0x17842E73,
  0xBBFEFF2F, 0x3C848B68, 0x31D7E0EC, 0x65228B39,
  0x37E49863, 0xE4C6D3B2, 0x3B0C849C, 0xF8424148,
  0x4BFE48F6, 0x1D59A5B1, 0x6BA06E6E, 0x12D1DA27,
  0xC5249A42, 0x1DEBD61B, 0x62EAB674, 0x6434EBC3,
  0xCC315E32, 0x220B3BAD, 0xD50BDC4C, 0x4E6C147F,
  0xEDD43D06, 0x80512BCB, 0xB42C07D4, 0x7349D215,
  0x3B70C4E5, 0xD7FDFCBF, 0xA36EA1A8, 0x5841B9E4,
  0x6E09A224, 0x5493D446, 0xC38D8CC0, 0xF1183746,
  0x90E7DF63, 0x3A8A4BFB, 0x3329B5EC, 0xE604B2B4,
  0xF37F4353, 0xC0869F4B, 0x9E17773D, 0xE68FEC45,
  0xE14904E0, 0xDEA45BF6, 0xCECF9918, 0xC85EA047,
  0x004C0AC6
};
static const uint32_t gbt329183e02[8] = {
  0x6B4B6D0E, 0x276691BD, 0x4A11BF72, 0xF4FB501A,
  0xE309FDAC, 0xB72FA6CC, 0x336E6656, 0x119ABD67
};
static const int gbt329183l02 = 211;
test_case gbt329183t02 = {
  (uint32_t*)&gbt329183m02,
  (uint32_t*)&gbt329183e02,
  gbt329183l02
};
tests[7] = gbt329183t02;

/*
 * GB/T 32918.3-2016 A.2 Example 3
 */
static const uint32_t gbt329183m03[56] = {
  0x47C82653, 0x4DC2F6F1, 0xFBF28728, 0xDD658F21,
  0xE174F481, 0x79ACEF29, 0x00F8B7F5, 0x66E40905,
  0xE4D1D0C3, 0xCA4C7F11, 0xBC8FF8CB, 0x3F4C02A7,
  0x8F108FA0, 0x98E51A66, 0x8487240F, 0x75E20F31,
  0x6B4B6D0E, 0x276691BD, 0x4A11BF72, 0xF4FB501A,
  0xE309FDAC, 0xB72FA6CC, 0x336E6656, 0x119ABD67,
  0x6CB56338, 0x16F4DD56, 0x0B1DEC45, 0x8310CBCC,
  0x6856C095, 0x05324A6D, 0x23150C40, 0x8F162BF0,
  0x0D6FCF62, 0xF1036C0A, 0x1B6DACCF, 0x57399223,
  0xA65F7D7B, 0xF2D9637E, 0x5BBBEB85, 0x7961BF1A,
  0x1799B2A2, 0xC7782953, 0x00D9A232, 0x5C686129,
  0xB8F2B533, 0x7B3DCF45, 0x14E8BBC1, 0x9D900EE5,
  0x54C9288C, 0x82733EFD, 0xF7808AE7, 0xF27D0E73,
  0x2F7C73A7, 0xD9AC98B7, 0xD8740A91, 0xD0DB3CF4
```

```
};
static const uint32_t gbt329183e03[8] = {
  0xFF49D95B, 0xD45FCE99, 0xED54A8AD, 0x7A709110,
  0x9F513944, 0x42916BD1, 0x54D1DE43, 0x79D97647
};
static const int gbt329183l03 = 224;
test_case gbt329183t03 = {
  (uint32_t*)&gbt329183m03,
  (uint32_t*)&gbt329183e03,
  gbt329183l03
};
tests[8] = gbt329183t03;

/*
 * GB/T 32918.3-2016 A.2 Example 4
 */
static const uint32_t gbt329183m04[17] = {
  0x022AF86E, 0xFE732CF1, 0x2AD0E09A, 0x1F2556CC,
  0x650D9CCC, 0xE3E24986, 0x6BBB5C68, 0x46A4C4A2,
  0x95FF49D9, 0x5BD45FCE, 0x99ED54A8, 0xAD7A7091,
  0x109F5139, 0x4442916B, 0xD154D1DE, 0x4379D976,
  0x00000047
};
static const uint32_t gbt329183e04[8] = {
  0x284C8F19, 0x8F141B50, 0x2E81250F, 0x1581C7E9,
  0xEEB4CA69, 0x90F9E02D, 0xF388B454, 0x71F5BC5C
};
static const int gbt329183l04 = 65;
test_case gbt329183t04 = {
  (uint32_t*)&gbt329183m04,
  (uint32_t*)&gbt329183e04,
  gbt329183l04
};
tests[9] = gbt329183t04;

/*
 * GB/T 32918.3-2016 A.2 Example 5
 */
static const uint32_t gbt329183m05[17] = {
  0x032AF86E, 0xFE732CF1, 0x2AD0E09A, 0x1F2556CC,
  0x650D9CCC, 0xE3E24986, 0x6BBB5C68, 0x46A4C4A2,
  0x95FF49D9, 0x5BD45FCE, 0x99ED54A8, 0xAD7A7091,
  0x109F5139, 0x4442916B, 0xD154D1DE, 0x4379D976,
  0x00000047
};
static const uint32_t gbt329183e05[8] = {
  0x23444DAF, 0x8ED75343, 0x66CB901C, 0x84B3BDBB,
  0x63504F40, 0x65C1116C, 0x91A4C006, 0x97E6CF7A
```

```
};
static const int gbt329183l05 = 65;
test_case gbt329183t05 = {
  (uint32_t*)&gbt329183m05,
  (uint32_t*)&gbt329183e05,
  gbt329183l05
};
tests[10] = gbt329183t05;


/*
 * GB/T 32918.3-2016 A.3 Example 2
 */
static const uint32_t gbt329183m07[55] = {
  0x00884249, 0x4C4C3435, 0x36405941, 0x484F4F2E,
  0x434F4D00, 0x00000000, 0x00000000, 0x00000000,
  0x00000000, 0x00000000, 0x00000000, 0x00000000,
  0x00000000, 0x00E78BCD, 0x09746C20, 0x2378A7E7,
  0x2B12BCE0, 0x0266B962, 0x7ECB0B5A, 0x25367AD1,
  0xAD4CC624, 0x2B00CDB9, 0xCA7F1E6B, 0x0441F658,
  0x343F4B10, 0x297C0EF9, 0xB6491082, 0x400A62E7,
  0xA7485735, 0xFADD013D, 0xE74DA659, 0x51C4D76D,
  0xC89220D5, 0xF7777A61, 0x1B1C38BA, 0xE260B175,
  0x951DC806, 0x0C2B3E00, 0x34297DD8, 0x3AB14D5B,
  0x393B6712, 0xF32B2F2E, 0x938D4690, 0xB095424B,
  0x89DA880C, 0x52D4A7D9, 0x0199BBF1, 0x1AC95A0E,
  0xA34BBD00, 0xCA50B93E, 0xC24ACB68, 0x335D20BA,
  0x5DCFE3B3, 0x3BDBD2B6, 0x0000002D
};
static const uint32_t gbt329183e07[8] = {
  0x557BAD30, 0xE183559A, 0xEEC3B225, 0x6E1C7C11,
  0xF870D22B, 0x165D015A, 0xCF9465B0, 0x9B87B527
};
static const int gbt329183l07 = 217;
test_case gbt329183t07 = {
  (uint32_t*)&gbt329183m07,
  (uint32_t*)&gbt329183e07,
  gbt329183l07
};
tests[12] = gbt329183t07;

/*
 * GB/T 32918.3-2016 A.3 Example 3
 */
static const uint32_t gbt329183m08[58] = {
  0x00DADD08, 0x7406221D, 0x657BC3FA, 0x79FF329B,
  0xB022E9CB, 0x7DDFCFCC, 0xFE277BE8, 0xCD4AE9B9,
  0x54ECF008, 0x0215977B, 0x2E5D6D61, 0xB98A9944,
```

```
  0x2F03E880, 0x3DC39E34, 0x9F8DCA56, 0x21A9ACDF,
  0x2B557BAD, 0x30E18355, 0x9AEEC3B2, 0x256E1C7C,
  0x11F870D2, 0x2B165D01, 0x5ACF9465, 0xB09B87B5,
  0x27018107, 0x6543ED19, 0x058C38B3, 0x13D73992,
  0x1D46B800, 0x94D961A1, 0x3673D4A5, 0xCF8C7159,
  0xE30401D8, 0xCFFF7CA2, 0x7A01A2E8, 0x8C186737,
  0x48FDE9A7, 0x4C1F9B45, 0x646ECA09, 0x97293C15,
  0xC34DD800, 0x2A4832B4, 0xDCD399BA, 0xAB3FFFE7,
  0xDD6CE6ED, 0x68CC43FF, 0xA5F2623B, 0x9BD04E46,
  0x8D322A2A, 0x0016599B, 0xB52ED9EA, 0xFAD01CFA,
  0x453CF305, 0x2ED60184, 0xD2EECFD4, 0x2B52DB74,
  0x110B984C, 0x00000023
};
static const uint32_t gbt329183e08[8] = {
  0xE05FE287, 0xB73B0CE6, 0x639524CD, 0x86694311,
  0x562914F4, 0xF6A34241, 0x01D885F8, 0x8B05369C
};
static const int gbt329183l08 = 229;
test_case gbt329183t08 = {
  (uint32_t*)&gbt329183m08,
  (uint32_t*)&gbt329183e08,
  gbt329183l08
};
tests[13] = gbt329183t08;

/*
 * GB/T 32918.3-2016 A.3 Example 4
 */
static const uint32_t gbt329183m09[17] = {
  0x0201F046, 0x4B1E8168, 0x4E5ED6EF, 0x281B5562,
  0x4EF46CAA, 0x3B2D3748, 0x4372D916, 0x10B69825,
  0x2CC9E05F, 0xE287B73B, 0x0CE66395, 0x24CD8669,
  0x43115629, 0x14F4F6A3, 0x424101D8, 0x85F88B05,
  0x00009C36
};
static const uint32_t gbt329183e09[8] = {
  0x4EB47D28, 0xAD3906D6, 0x244D01E0, 0xF6AEC73B,
  0x0B51DE15, 0x74C13798, 0x184E4833, 0xDBAE295A
};
static const int gbt329183l09 = 66;
test_case gbt329183t09 = {
  (uint32_t*)&gbt329183m09,
  (uint32_t*)&gbt329183e09,
  gbt329183l09
};
tests[14] = gbt329183t09;

/*
```

```
 * GB/T 32918.3-2016 A.3 Example 5
 */
static const uint32_t gbt329183m10[17] = {
  0x0301F046, 0x4B1E8168, 0x4E5ED6EF, 0x281B5562,
  0x4EF46CAA, 0x3B2D3748, 0x4372D916, 0x10B69825,
  0x2CC9E05F, 0xE287B73B, 0x0CE66395, 0x24CD8669,
  0x43115629, 0x14F4F6A3, 0x424101D8, 0x85F88B05,
  0x00009C36
};
static const uint32_t gbt329183e10[8] = {
  0x588AA670, 0x64F24DC2, 0x7CCAA1FA, 0xB7E27DFF,
  0x811D500A, 0xD7EF2FB8, 0xF69DDF48, 0xCC0FECB7
};
static const int gbt329183l10 = 66;
test_case gbt329183t10 = {
  (uint32_t*)&gbt329183m10,
  (uint32_t*)&gbt329183e10,
  gbt329183l10
};
tests[15] = gbt329183t10;

/*
 * GB/T 32918.4-2016 A.2 Example 1
 */
static const uint32_t gbt329184m01[17] = {
  0x57E7B636, 0x23FAE5F0, 0x8CDA468E, 0x872A20AF,
  0xA03DED41, 0xBF140377, 0x656E6372, 0x79707469,
  0x6F6E2073, 0x74616E64, 0x6172640E, 0x040DC83A,
  0xF31A6799, 0x1F2B01EB, 0xF9EFD888, 0x1F0A0493,
  0x00030600
};
static const uint32_t gbt329184e01[8] = {
  0x6AFB3BCE, 0xBD76F82B, 0x252CE5EB, 0x25B57996,
  0x86902B8C, 0xF2FD8753, 0x6E55EF76, 0x03B09E7C
};
static const int gbt329184l01 = 67;
test_case gbt329184t01 = {
  (uint32_t*)&gbt329184m01,
  (uint32_t*)&gbt329184e01,
  gbt329184l01
};
tests[16] = gbt329184t01;

/*
 * GB/T 32918.4-2016 A.2 Example 2
 */
static const uint32_t gbt329184m02[21] = {
  0x64D20D27, 0xD0632957, 0xF8028C1E, 0x024F6B02,
```

```
  0xEDF23102, 0xA566C932, 0xAE8BD613, 0xA8E865FE,
  0x656E6372, 0x79707469, 0x6F6E2073, 0x74616E64,
  0x61726458, 0xD225ECA7, 0x84AE300A, 0x81A2D482,
  0x81A828E1, 0xCEDF11C4, 0x21909984, 0x02653750,
  0x0078BF77
};
static const uint32_t gbt329184e02[8] = {
  0x9C3D7360, 0xC30156FA, 0xB7C80A02, 0x76712DA9,
  0xD8094A63, 0x4B766D3A, 0x285E0748, 0x0653426D
};
static const int gbt329184l02 = 83;
test_case gbt329184t02 = {
  (uint32_t*)&gbt329184m02,
  (uint32_t*)&gbt329184e02,
  gbt329184l02
};
tests[17] = gbt329184t02;

/*
 * GB/T 32918.4-2016 A.3 Example 1
 */
static const uint32_t gbt329184m03[18] = {
  0x01C6271B, 0x31F6BE39, 0x6A4166C0, 0x616CF4A8,
  0xACDA5BEF, 0x4DCBF2DD, 0x42656E63, 0x72797074,
  0x696F6E20, 0x7374616E, 0x64617264, 0x0147AF35,
  0xDFA1BFE2, 0xF161521B, 0xCF59BAB8, 0x3564868D,
  0x92958817, 0x00000035
};
static const uint32_t gbt329184e03[8] = {
  0xF0A41F6F, 0x48AC723C, 0xECFC4B76, 0x7299A5E2,
  0x5C064167, 0x9FBD2D4D, 0x20E9FFD5, 0xB9F0DAB8
};
static const int gbt329184l03 = 69;
test_case gbt329184t03 = {
  (uint32_t*)&gbt329184m03,
  (uint32_t*)&gbt329184e03,
  gbt329184l03
};
tests[18] = gbt329184t03;

/*
 * GB/T 32918.4-2016 A.3 Example 2
 */
static const uint32_t gbt329184m04[22] = {
  0x0083E628, 0xCF701EE3, 0x141E8873, 0xFE55936A,
  0xDF24963F, 0x5DC9C648, 0x0566C80F, 0x8A1D8CC5,
  0x1B656E63, 0x72797074, 0x696F6E20, 0x7374616E,
  0x64617264, 0x01524C64, 0x7F0C0412, 0xDEFD468B,
```

```
      0xDA3AE0E5, 0xA80FCC8F, 0x5C990FEE, 0x11602929,
      0x232DCD9F, 0x00000036
    };
    static const uint32_t gbt329184e04[8] = {
      0x73A48625, 0xD3758FA3, 0x7B3EAB80, 0xE9CFCABA,
      0x665E3199, 0xEA15A1FA, 0x8189D96F, 0x579125E4
    };
    static const int gbt329184l04 = 85;
    test_case gbt329184t04 = {
      (uint32_t*)&gbt329184m04,
      (uint32_t*)&gbt329184e04,
      gbt329184l04
    };
    tests[19] = gbt329184t04;

    for(i = 0; i < 20; i++)
    {
      debug_print("\n\n= Running Example[%i]\n", i);
      printf("sm3_example[%2i]: %s\n", i,
        sm3_run_example(tests[i]) ?  "FAIL" : "PASS");
      //printf("\n\n\n\n\n\n\n\n\n\n");
      debug_print("\n\n= Running Example[%i] Done\n", i);
    }

    return 0;
  }
  <CODE ENDS>
```

C.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 SM3PRINT_H
#define SM3PRINT_H

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

#include <inttypes.h>

void print_bytes(unsigned* buf, int n);
void print_block(unsigned* buf, int n);
void print_af(int i, uint32_t A, uint32_t B, uint32_t C, uint32_t D,
    uint32_t E, uint32_t F, uint32_t G, uint32_t H);
void print_hash(unsigned* buf);

#endif
<CODE ENDS>
```

"print.c"

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

void print_bytes(unsigned *buf, int n)
{
  uint8_t *ptr = (uint8_t*)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");
}

void print_block(unsigned *buf, int n)
{
  print_bytes(buf, n * 4);
}
```

```
   void print_af(int i,
       uint32_t A, uint32_t B,
       uint32_t C, uint32_t D,
       uint32_t E, uint32_t F,
       uint32_t G, uint32_t H)
   {
       if (i % 10 == 0) {
         debug_print("\n");
         debug_print("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
             " j = %2d "
             "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n", i);
       }
       debug_print("%.8X ", (unsigned)A);
       debug_print("%.8X ", (unsigned)B);
       debug_print("%.8X ", (unsigned)C);
       debug_print("%.8X ", (unsigned)D);
       debug_print("%.8X ", (unsigned)E);
       debug_print("%.8X ", (unsigned)F);
       debug_print("%.8X ", (unsigned)G);
       debug_print("%.8X",  (unsigned)H);
       debug_print("\n");
   }

   void print_hash(unsigned *buf)
   {
     print_block(buf, 8);
   }
   <CODE ENDS>
```

Appendix D. Acknowledgements

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

- Erick Borsboom for the lengthy review of this document and example verification;
- Jack Lloyd and Daniel Wyatt of the Ribose RNP team for their input and implementation.

Authors' Addresses

Sean Shen
Computer Network Information Center, Chinese Academy of Sciences
4 Zhongguancun South Fourth Street, Zhongguancun, Haidian District
Beijing 100190
People's Republic of China

Email: sean.s.shen@gmail.com
URI: http://www.cnic.cn

Xiaodong Lee
Institute of Computing Technology, Chinese Academy of Sciences
6 Kexueyuan South Street, Haidian District
Beijing 100190
People's Republic of China

Email: xl@ict.ac.cn
URI: http://www.ict.ac.cn

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

Paul Y. Yang
BaishanCloud
Building 16-3, Baitasan Street
Shenyang, Liaoning 110000
People's Republic of China

Email: yang.yang@baishancloud.com
URI: https://www.baishancloud.com