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

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents
Table of Contents

1. Introduction ......................................................... 4
   1.1. Purpose ....................................................... 4
   1.2. History ...................................................... 5
   1.3. Applications .................................................. 5
2. Terms and Definitions .................................................. 5
3. Symbols And Abbreviations ............................................... 6
   3.1. Operators ...................................................... 6
   3.2. Usage ........................................................ 7
4. Primitives And Functions ................................................ 8
   4.1. Initialization Vector IV ........................................ 8
   4.2. Constants T_j .................................................. 8
   4.3. Boolean Functions FF_j and GG_j ................................ 9
   4.4. Permutation Functions P_0 and P_1 ................................ 9
5. Algorithm ............................................................ 9
   5.1. Overview ....................................................... 9
   5.2. Padding PAD .................................................... 9
   5.3. Iterative Hashing ............................................... 10
      5.3.1. Iterative Compression Process .............................. 10
      5.3.2. Message Expansion Function ME ............................... 11
      5.3.3. Compression Function CF .................................... 12
      5.3.4. Hash Value ................................................ 13
6. Design Rationale ....................................................... 13
   6.1. General Design Principles ...................................... 14
   6.2. Compression Function Design .................................... 14
      6.2.1. Compression Function Design Principles ...................... 14
      6.2.2. Selection Of Permutation Function Parameters ............. 15
      6.2.3. Selection Of Boolean Functions .............................. 15
      6.2.4. Selection Of Rotational Constants R and R’ .............. 15
      6.2.5. Selection Of Rotational Constants S and S’ .............. 16
      6.2.6. Selection Of Addition Constants ............................ 16
   6.3. Message Expansion Design ....................................... 17
7. Cryptanalysis ......................................................... 17
   7.1. Analysis ........................................................ 17
      7.1.1. Differential Analysis ...................................... 17
      7.1.2. Preimage Attacks .......................................... 18
      7.1.3. Distinguishing Attacks .................................... 19
   7.2. Comparison with Other Algorithms ................................ 20
8. Object Identifier ........................................................ 22
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 \( 0 \leq j \leq 15 \):

\( T_j = 79cc4519 \)

When \( 16 \leq j \leq 63 \):

\( T_j = 7a879d8a \)

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:

\[ FF\_j(X, Y, Z) = X \text{xor} Y \text{xor} Z \]
\[ GG\_j(X, Y, Z) = X \text{xor} Y \text{xor} Z \]

When 16 <= j <= 63:

\[ FF\_j(X, Y, Z) = (X \text{ and } Y) \text{ or } (X \text{ and } Z) \text{ or } (Y \text{ and } Z) \]
\[ GG\_j(X, Y, Z) = (X \text{ and } Y) \text{ or } (\neg X \text{ and } Z) \]

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

\[ P\_0(X) = X \text{xor} (X \ll 9) \text{xor} (X \ll 17) \]
\[ P\_1(X) = X \text{xor} (X \ll 15) \text{xor} (X \ll 23) \]

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 \( L = \text{num2str}(l, 64) \).

Inputs:
- m, the original message m of length l bits.

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

m' is defined as follows:

\[
\begin{align*}
  l &= \text{bitlen}(m) \\
  L &= \text{num2str}(l, 64) \\
  k &= 512 - (((l \mod 512) + 1 + 64) \mod 512) \\
  K &= \text{num2str}(0, k) \\
  m' &= m \parallel 1 \parallel K \parallel L
\end{align*}
\]

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

```
#1 m            #2  #3       #4 L
________________________  _  _____   _________
/                        \\ | /     \\ /         \\    \\
 m' = 01100001 01100010 01100011 1 00...00 00...011000
________________________/   \\_____/ \_________/    \\
24-bits            423-bits   64-bits
\______________________________________________/
512-bits
```

5.3. Iterative Hashing

5.3.1. Iterative Compression Process

Inputs:
- 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.
Internet-Draft           SM3 Cryptographic Hash            November 2017

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.
32-bits

<table>
<thead>
<tr>
<th>R</th>
<th>R'</th>
<th>(R + R')</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 10 20 30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

16-bits

<table>
<thead>
<tr>
<th>S</th>
<th>S'</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6</td>
<td></td>
</tr>
<tr>
<td>0 10</td>
<td></td>
</tr>
</tbody>
</table>

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.

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

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], Stroibog [GOSTR.34.11.2012] and SHA-3 [NIST.FIPS.202] are shown in Table 2.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Attack Type</th>
<th>Steps / Rounds</th>
<th>%</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM3</td>
<td>Collision</td>
<td>20</td>
<td>31</td>
<td>[SM3-Coll]</td>
</tr>
<tr>
<td>SM3</td>
<td>Preimage</td>
<td>30</td>
<td>47</td>
<td>[SM3-Pre]</td>
</tr>
<tr>
<td>SM3</td>
<td>Distinguisher</td>
<td>37</td>
<td>58</td>
<td>[SM3-Boomerang2]</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Collision</td>
<td>80</td>
<td>100</td>
<td>[SHA1-Coll]</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Preimage</td>
<td>62</td>
<td>77.5</td>
<td>[SHA1-HDPre]</td>
</tr>
<tr>
<td>RIPEMD-128</td>
<td>Collision</td>
<td>40</td>
<td>62.5</td>
<td>[RIPE128-Coll]</td>
</tr>
<tr>
<td>RIPEMD-128</td>
<td>Preimage</td>
<td>36</td>
<td>62.5</td>
<td>[RIPE128-Pre]</td>
</tr>
<tr>
<td>RIPEMD-128</td>
<td>Distinguisher</td>
<td>64</td>
<td>100</td>
<td>[RIPE128-Crypt]</td>
</tr>
<tr>
<td>RIPEMD-160</td>
<td>Preimage</td>
<td>34</td>
<td>53.12</td>
<td>[RIPE160-Pre]</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Collision</td>
<td>31</td>
<td>48.4</td>
<td>[SHA256-Coll]</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Preimage</td>
<td>45</td>
<td>70.3</td>
<td>[SHA256-Pre]</td>
</tr>
<tr>
<td>SHA-256</td>
<td>Distinguisher</td>
<td>47</td>
<td>73.4</td>
<td>[SHA256-Diff]</td>
</tr>
<tr>
<td>Whirlpool</td>
<td>Collision</td>
<td>8</td>
<td>80</td>
<td>[WP-PC]</td>
</tr>
<tr>
<td>Whirlpool</td>
<td>Preimage</td>
<td>6</td>
<td>60</td>
<td>[WP-PC]</td>
</tr>
<tr>
<td>Whirlpool</td>
<td>Distinguisher</td>
<td>10</td>
<td>100</td>
<td>[WP-Rebound]</td>
</tr>
<tr>
<td>Stribog-256</td>
<td>Collision</td>
<td>6.5</td>
<td>54.2</td>
<td>[ST-Pre]</td>
</tr>
<tr>
<td>Stribog-512</td>
<td>Collision</td>
<td>7.5</td>
<td>62.5</td>
<td>[ST-Pre]</td>
</tr>
<tr>
<td>Stribog-512</td>
<td>Preimage</td>
<td>6</td>
<td>50</td>
<td>[ST-Pre]</td>
</tr>
<tr>
<td>Stribog-512</td>
<td>Distinguisher</td>
<td>6</td>
<td>50</td>
<td>[ST-Boom]</td>
</tr>
<tr>
<td>SHA3-224</td>
<td>Collision</td>
<td>5</td>
<td>20.8</td>
<td>[SHA3-SLin]</td>
</tr>
<tr>
<td>SHA3-256</td>
<td>Collision</td>
<td>5</td>
<td>20.8</td>
<td>[SHA3-Coll]</td>
</tr>
<tr>
<td>SHA3-256</td>
<td>Preimage</td>
<td>4</td>
<td>16.7</td>
<td>[SHA3-Rot]</td>
</tr>
<tr>
<td>SHA3-512</td>
<td>Collision</td>
<td>3</td>
<td>12.5</td>
<td>[SHA3-Coll]</td>
</tr>
<tr>
<td>SHAKE-128</td>
<td>Collision</td>
<td>5</td>
<td>20.8</td>
<td>[SHA3-Coll12]</td>
</tr>
<tr>
<td>Keccak-f</td>
<td>Distinguisher</td>
<td>24</td>
<td>100</td>
<td>[KEKKAC-ZSD]</td>
</tr>
</tbody>
</table>

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.

o 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]

[ISO.IEC.10118-3]


11.2. Informative References

[ADDEND.DEPS]

[BOTAN]


[MD4-Coll]

[MD5-Coll]

[NBS-Crypt]

[NIST.FIPS.180-1]

[NIST.FIPS.180-2]

[NIST.FIPS.180-4]

[NIST.FIPS.202]


[SHA1-Char]

[SHA1-Coll]

[SHA1-Coll2]

[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]

[SHA1-Pre]

[SHA2-Char]

[SHA2-Pre]

[SHA256-Coll]


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 00000000 00000000 00000000 00000018

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

The message after expansion is shown below.

\[ W_0 \ W_1 \ldots \ W_{67} = \]

\[ 61626380 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000018 \]

\[ 9092e200 \ 00000000 \ 000c0606 \ 719c70ed \ 00000000 \ 8001801f \ 939f7da9 \ 00000000 \ 2c6fa1f9 \ adaef14 \ 00000000 \ 0001801e \ 9a965f89 \ 49710048 \ 23ce86a1 \ b2d12f1b \]

\[ e1dae338 \ f8061807 \ 055d68be \ 86cfd481 \ 1f447d83 \ d9023dbf \ 185898e0 \ e0061807 \]

\[ 050df55c \ cde0104c \ a5b9c955 \ a7df0184 \ 6e46cd08 \ e3babdf8 \ 70ca422 \ 0353af50 \]

\[ a92docal1 \ 5f33cfd2 \ e166f6e9 \ f70fe941 \ ca5462dc \ 85a90152 \ 76af6296 \ c922b9b2 \]

\[ 68378cf5 \ 97585344 \ 09008723 \ 86faee74 \ 2ab908b0 \ 4a64bc50 \ 8646e808 \ f07e6590 \]

\[ 325c8f78 \ accb8011 \ e11db9dd \ b99c0545 \]

\[ W'_0 \ W'_1 \ldots \ W'_{63} = \]

\[ 61626380 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000000 \ 00000018 \]

\[ 9092e200 \ 00000000 \ 000c0606 \ 719c70ed \ 2c6fa1f9 \ adaef14 \ 00000000 \ 0001801e \ 9a965f89 \ 49710048 \ 23ce86a1 \ b2d12f1b \]

\[ e1dae338 \ f8061807 \ 055d68be \ 86cfd481 \ 1f447d83 \ d9023dbf \ 185898e0 \ e0061807 \]

\[ 050df55c \ cde0104c \ a5b9c955 \ a7df0184 \ 6e46cd08 \ e3babdf8 \ 70ca422 \ 0353af50 \]

\[ a92docal1 \ 5f33cfd2 \ e166f6e9 \ f70fe941 \ ca5462dc \ 85a90152 \ 76af6296 \ c922b9b2 \]

\[ 68378cf5 \ 97585344 \ 09008723 \ 86faee74 \ 2ab908b0 \ 4a64bc50 \ 8646e808 \ f07e6590 \]

\[ 325c8f78 \ accb8011 \ e11db9dd \ b99c0545 \]

A.1.4. Intermediate Values During Iterative Compression

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial value</td>
<td>7380166f</td>
<td>4914b2b9</td>
<td>172442d7</td>
<td>da8a0600</td>
<td>a96f30bc</td>
<td>163138aa</td>
<td>e38de4ed</td>
</tr>
</tbody>
</table>

j = 0

| b9edc12b | 7380166f | 29657292 | 172442d7 | b2ad29f4 | a96f30bc | c550b189 | e38de4ed | ea52428c | b9edc12b | 002cdee7 | 29657292 | ac353a23 | b2ad29f4 | 85e54b79 | c550b189 |
Internet-Draft       SM3 Cryptographic Hash       November 2017

48dcbbc62 d4f4efe3 c3bbc4ab e6c7f233 6f49c7bb 5cfba85a 667a2ced d6de1815
8237b8a0 48dcbbc62 e9dfc7a9 c3bbc4ab d89d2711 6f49c7bb 42d2e7dd 667a2ced
d8685939 8237b8a0 b978c491 e9dfc7a9 8ee87df5 d89d2711 3d2b7a4e 42d2e7dd
d2090a86 d8685939 6f714104 b978c491 2e533625 8ee87df5 388ec4e9 3d2b7a4e
e51076b3 d2090a86 d0b273b0 6f714104 d9f89e61 2e533625 efac7743 388ec4e9
47c5be50 e51076b3 12050da4 d0b273b0 3567734e 9d8f9e61 b2c79299 efac7743
Abddbbdc8 47c5be50 206ed7ca 12050da4 3d2b7a4e 3567734e f30e6cfc f2d79299
bd708003 abddbbdc8 8b7ca08f 206ed7ca 93499bc0 3d2b7a4e 9a71ab3b f30e6cfc

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 50 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
15e2f5d3 bd708003 bb7b9157 8b7ca08f c3956c3f 93499bc0 e889e9f6 9a71ab3b
13826486 15e2f5d3 e10077a bb7b9157 cd09a51c c3956c3f 5e049a4a e889e9f6
4a00ed2f 13826486 c5eba62b e10077a 0741f675 cd09a51c 61fe1cb9 5e049a4a
f4412e82 4a00ed2f 04c90c27 c5eba62b 7249907c 0741f675 28e6684d 61fe1cb9
549db4b7 f4412e82 01da5e94 04c90c27 f6bc15ed 7429807c b3a83a0f 28e6684d
22a79585 549db4b7 82d505e8 01da5e94 9d4da6b2 6cbe15ed 03e3a149 b3a83a0f
30245b78 22a79585 3b696ea9 82d505e8 68048c82 9d4da6b2 af6f95e0 03e3a149
5d981349 30245b78 14fb0764 3b696ea9 9f20a45 68048c82 83d4e6ad af6f95e0
3c6d29a9 5d98134f 48bf6060 14fb0764 3b696ea9 9f20a45 68048c82 83d4e6ad
dddb0a26a c36d29a9 30629ceb 48bf6060 589f7d5c 14fb0764 69d8a15 641b7402

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ j = 60 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
7103d471 ddb0a26a ac533587 30629ceb b7c7d7c7 589f7d5c 82b0a457 6d97a915
5e636b4b 7103d471 6144d5bb ac533587 09c9d5e5 14d5c7f6 eae24c4f 3b20a7d8
2bfa5f60 5e636b4b 069ae2e2 6144d5bb 4ac3c0f8 09c9d5e5 3f0a6ae eae24c4f
1547e69b 2bfa5f60 c6696bc 069ae2e2 e8084f3b 4ac3c0f8 c0a0e6e3 3f0a6ae

A.1.5. Hash Value

66c7f0f4 62eeed9d d1f2d46b dc104e2 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

A.2.2. Padded Message

The message after padding is shown below.

A.2.2.1. Message Block 1

A.2.2.1.1. Expanded Message

\[ W_0 \ W_1 \ldots \ W_{67} : \]

\[ \begin{align*}
7380166f & \quad 7380166f \\
4914b2b9 & \quad 4914b2b9 \\
172442d7 & \quad 172442d7 \\
da8a0600 & \quad da8a0600 \\
a96f30bc & \quad a96f30bc \\
163138aa & \quad 163138aa \\
e38de44d & \quad e38de44d \\
b0fb0e4e & \quad b0fb0e4e \\
7380166f & \quad 7380166f \\
29657292 & \quad 29657292 \\
172442d7 & \quad 172442d7 \\
b2e561d0 & \quad b2e561d0 \\
a96f30bc & \quad a96f30bc \\
c550b189 & \quad c550b189 \\
e38de44d & \quad e38de44d \\
b31cecd3 & \quad b31cecd3 \\
588b5dab & \quad 588b5dab \\
002cdee7 & \quad 002cdee7 \\
29657292 & \quad 29657292 \\
887cdf53 & \quad 887cdf53 \\
b2e561d0 & \quad b2e561d0 \\
85e54b79 & \quad 85e54b79 \\
c550b189 & \quad c550b189 \\
087b31df & \quad 087b31df \\
b31cecd3 & \quad b31cecd3 \\
16bb5d61 & \quad 16bb5d61 \\
002cdee7 & \quad 002cdee7 \\
5234344f & \quad 5234344f \\
887cdf53 & \quad 887cdf53 \\
0e85972b & \quad 0e85972b \\
85e54b79 & \quad 85e54b79 \\
1744bb12 & \quad 1744bb12 \\
087b31df & \quad 087b31df \\
39da7665 & \quad 39da7665 \\
16bb5d61 & \quad 16bb5d61 \\
16372ca6 & \quad 16372ca6 \\
5234344f & \quad 5234344f \\a9c43e6 & \quad fa9c43e6 \\
0e85972b & \quad 0e85972b \\
dca06de5 & \quad dca06de5 \\
1744bb12 & \quad 1744bb12 \\
f663be10 & \quad f663be10 \\
39da7665 & \quad 39da7665 \\
f7bc113c & \quad f7bc113c \\
16372ca6 & \quad 16372ca6 \\
a27a91a1 & \quad a27a91a1 \\a9c43e6 & \quad fa9c43e6 \\
8eb847a3 & \quad 8eb847a3 \\
dca06de5 & \quad dca06de5 \\
8916242e & \quad 8916242e \\
f663be10 & \quad f663be10 \\
9fe64fb1 & \quad 9fe64fb1 \\
f7bc113c & \quad f7bc113c \\
6530b1b9 & \quad 6530b1b9 \\
a27a91a1 & \quad a27a91a1 \\
0e0f1218 & \quad 0e0f1218 \\
e8e847a3 & \quad e8e847a3 \\
40dcbcb9 & \quad 40dcbcb9 \\
8916242e & \quad 8916242e \\
57e5fc4e & \quad 57e5fc4e \\
9fe64fb1 & \quad 9fe64fb1 \\
89e7bde0 & \quad 89e7bde0 \\
6530b1b9 & \quad 6530b1b9 \\
da8a3827 & \quad da8a3827 \\
0e0f1218 & \quad 0e0f1218 \\
708f471d & \quad 708f471d \\
40dcbcb9 & \quad 40dcbcb9 \\
55eb8591 & \quad 55eb8591 \\
57e5fc4e & \quad 57e5fc4e \\
7d88ff32 & \quad 7d88ff32 \\
89e7bde0 & \quad 89e7bde0 \\
6e12c163 & \quad 6e12c163 \\
da8a3827 & \quad da8a3827 \\
1e24301c & \quad 1e24301c \\
708f471d & \quad 708f471d \\
c26a1a88 & \quad c26a1a88 \\
55eb8591 & \quad 55eb8591 \\
e272bf2f & \quad e272bf2f \\
7d88ff32 & \quad 7d88ff32 \\
f7578117 & \quad f7578117 \\
6e12c163 & \quad 6e12c163 \\
50704f5b & \quad 50704f5b \\
c26a1a88 & \quad c26a1a88 \\
2c8aa5f5c & \quad 2c8aa5f5c \\
e272bf2f & \quad e272bf2f
\end{align*} \]

A.2.2.1.2. Intermediate Values During Iterative Compression

\[ \begin{array}{cccccccc}
A & B & C & D & E & F & G & H \\
\hline
7380166f & 4914b2b9 & 172442d7 & da8a0600 & a96f30bc & 163138aa & e38de44d & b0fb0e4e \\
588b5dab & 7380166f & 29657292 & 172442d7 & b2e561d0 & a96f30bc & c550b189 & e38de44d \\
b31cecd3 & 588b5dab & 002cdee7 & 29657292 & 887cdf53 & b2e561d0 & 85e54b79 & c550b189 \\
087b31df & b31cecd3 & 16bb5d61 & 002cdee7 & 5234344f & 887cdf53 & 0e85972b & 85e54b79 \\
1744bb12 & 087b31df & 39da7665 & 16bb5d61 & 16372ca6 & 5234344f & fa9c43e6 & 0e85972b \\
dca06de5 & 1744bb12 & f663be10 & 39da7665 & f7bc113c & 16372ca6 & a27a91a1 & fa9c43e6 \\
8eb847a3 & dca06de5 & 8916242e & f663be10 & 9fe64fb1 & f7bc113c & 6530b1b9 & a27a91a1 \\
0e0f1218 & 8eb847a3 & 40dcbcb9 & 8916242e & 57e5fc4e & 9fe64fb1 & 89e7bde0 & 6530b1b9 \\
da8a3827 & 0e0f1218 & 708f471d & 40dcbcb9 & 55eb8591 & 57e5fc4e & 7d88ff32 & 89e7bde0 \\
6e12c163 & da8a3827 & 1e24301c & 708f471d & c26a1a88 & 55eb8591 & e272bf2f & 7d88ff32 \\
f7578117 & 6e12c163 & 50704f5b & 1e24301c & 3433dd28 & c26a1a88 & 2c8aa5f5c & e272bf2f
\end{array} \]
bc497c66 f7578117 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 bc497c66 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 f7578117 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
ecc59168 af022fee 2582c6dc 50704f5b 4f85c749 3433dd28 2c8aaf5c
6327315 231d84bd 9f532735 c933a698 deefae1d 108149de 71c6ef02 2f1924a3 ea3339fc
6a203212 231d84bd a64e6b3e c933a698 90c31af9 108149de 78138e37 2f1924a3
175c3b57 6a203212 3b097a46 a64e6b3e 508f82d2 90c31af9 4ef0840a 78138e37
cdcbab5d 175c3b57 40642d4d 3b097a46 b5a2f2ff 508f82d2 d7cc8618 4ef0840a
7dd941f8 cdcbab5d b876aae2 40642d4d a541cb9b b5a2f2ff 1692847c d7cc8618
eaf5f3e7 7dd941f8 9757ab9b b876aae2 9124d1e7 a541cb9b 97dadd17 1692847c
f731oa83 eaf5f3e7 b283f0fb 9757ab9b b43da5e9 9124d1e7 730c672f 97dadd17
f8441d7e f731oa83 ea9e7dd5 b283f0fb cf194872 b43da5e9 5cdd2a0e 97dadd17
270dce67 f8441d7e 621507ee 964015e3 7564b6c0 68ef7357 4c6499d3 4c6499d3
dee6aed1 1b9ccee4 883afdf0 0fac4cad 964015e3 b603ab25 349678ca
32418d74 1b9ccee4 254d8158 1b9ccee4 3f717698 0fac4cad af1cb200 b603ab25
9c89b505 32418d74 b33cd637 254d8158 38766aef 3f717698 656876d2 af1cb200
3c60352a 9c89b505 831ae864 b33cd637 8aed993b 38766aef b41cf8bf 656876d2
2a116c70 3c60352a 136ab039 831ae864 476048d4 8aed993b 55f9c3b3 b41cf8bf
a0c7c66f 2a116c70 c06a5478 136ab039 b47a7dc5 476048d4 c9dc576e 5f9c3b3
b7e5f833 a0c7c66f 22de8e054 c06a5478 a3537a9 b47a7dc5 46a23b02 c9dc576e
79baf4ca b7e5f833 8fd8c4f1 22de8e054 9455b731 3a3537a9 ee2ada3d 46a23b02
ad5b0bcbf 79baf4ca cbe1676f 8f8c4df1 289d35e0 9455b731 bd49d1da9 ee2ada3d
a167bd76 ae167bd76 759e94f3 cbe1676f da27726b 289d35e0 b98ca2ad bd49d1d9
2ccc1878 a167bd76 b6179f5a 75e94f3 7ed6d43b da27276b af0144e9 b98ca2ad
610c6084 2ccc1878 cf7aed42 b6179f5a 9da32cab 7ed6d43b 3b5ed139 af0144e9
a40209fe 610c6084 9830f509 cf7aed42 7d483846 9da32cab a1dbf6f6 b5ed1d9
6fa376a2 a40209fe 18c108c2 9830f509 12a851cf 7d483846 655ced19 a1dbf6f6
53f9fffc5 6fa376a2 0413fd48 18c108c2 c3d3327b 12a851cf c233ea41 655ced19
4f600bd5 53f9fffc5 46ed4d4f 0413fd48 3f3ace7e c3d3327b 8e789542 c233ea41
6e89a7fb 4f600bd5 3f3f8a77 46ed4d4f 17394ca0 3f3ace7e 93de1e99 8e789542
f9f3c8b6 6e89a7fb c177a9e 3f3f8a77 4a9e59f4 17394ca0 f379e57 93de1e99
fa8e6731 fef3c8b6 134ff6dd c177a9e 7d9e1966 4a9e59f4 6500b9ca 3f3f8a77
08a826c3 fa8e6731 e7962f6d 134ff6dd ebf9a0cc 7d9e1966 ca7a5f42 6500b9ca
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
80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
80400000 00000000 01008080 10005000 00000000 002002a0 ac545c04 00000000
8052b53 a0003000 00000000 02020080 a4515804 20200040 51609838 30005701
a0002000 002000aa 6ad525d0 0a0e0216 b0f52042 fa7073b0 20000000 008200a8
7a542590 22a20044 d56ebed2 82005771 8a202240 b42826aa eaf84e59 4898eaf9
807283 04e6775fa a3e0e0a 0828488a 23b45a5d 682a22c4 8d6d0615 38300a7e
e96260e5 2b60c020 502ed531 9e878cb9 218c38f8 dca3e0a7 e9e0c461 8c3e3831 44aa4a28 dc60a38b 518300f7
```

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

```
80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
80000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
80404000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
8052b53 a0003000 02020000 a4515804 20200040 51609838 30005701
a0002000 00000000 6ad525d0 0a0e0216 b0f52042 fa7073b0 20000000 008200a8
7a542590 22a20044 d56ebed2 82005771 8a202240 b42826aa eaf84e59 4898eaf9
807283 04e6775fa a3e0e0a 0828488a 23b45a5d 682a22c4 8d6d0615 38300a7e
e96260e5 2b60c020 502ed531 9e878cb9 218c38f8 dca3e0a7 e9e0c461 8c3e3831 44aa4a28 dc60a38b 518300f7
```
### A.2.2.2.2. Intermediate Values During Iterative Compression

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td>5950de81</td>
<td>468664eb</td>
<td>42fd4c86</td>
<td>1e7ca00a</td>
<td>c0a5910b</td>
<td>ae9a55ea</td>
<td>1adb8d17</td>
<td>763ca222</td>
</tr>
<tr>
<td>1cc66027</td>
<td>5950de81</td>
<td>0cc9d68d</td>
<td>42fd4c86</td>
<td>24fe81a1</td>
<td>c0a5910b</td>
<td>af5574d2</td>
<td>1adb8d17</td>
</tr>
<tr>
<td>920d5d4d</td>
<td>b1aacb3f</td>
<td>32e6496e</td>
<td>8cc04e39</td>
<td>c6c863a3</td>
<td>4c7cbb59</td>
<td>cbsd30db9</td>
<td>0d927f4</td>
</tr>
<tr>
<td>0162191</td>
<td>920d5d4d</td>
<td>55967f63</td>
<td>32e6496e</td>
<td>dbcb73dd</td>
<td>c6c863a3</td>
<td>dac2e6a</td>
<td>cb30db9</td>
</tr>
<tr>
<td>cbfddbb7</td>
<td>03162191</td>
<td>1acb9b24</td>
<td>55967f63</td>
<td>6a6eaaab</td>
<td>dbcb73dd</td>
<td>1de3643</td>
<td>dac2e6a</td>
</tr>
</tbody>
</table>

| j = 0 |

| df0f77ed | 502a9be2 | ff301aee | 80c727bf | c8b999f7 | befc3eda | 02cc4e85 | 46acec25 |
| b80c2081 | df0f77ed | 5537c4a0 | ff301aee | 3a05da38 | c8b999f7 | 02cc4e85 | 46acec25 |
| 5b3aaa5 | b80c2081 | 1eefdbee | 5537c4a0 | eefbf718 | 3a05da38 | cfbe45cc | 02cc4e85 |
| 0f71b5e4 | 5b3aaa5 | 785003ee | 1eefdbee | f3fbf696 | eefbf718 | d1c1d02e | cfbe45cc |
| 141bc167 | 0f71b5e4 | 77555a6b | 78500371 | leefdbee | f3fbf696 | d1c1d02e | cfbe45cc |
| f1b5448b | 141bc167 | e30bc81e | 77555a6b | 3a05da38 | c8b999f7 | 55967f63 | 6a6eaaab |
| aac2adc7 | f1b5448b | 3963ece8 | e30bc81e | 3d03e81b | indianred | 6a6eaaab | e996d68b |
| 3d2dfda3 | aac2adc7 | 6e959be4 | 0a8915e3 | 0b7fff8f | 3d03e81b | 6a6eaaab | e996d68b |
| 15bab3e6 | 3d2dfda3 | 945b9755 | 6e959be4 | 78500371 | leefdbee | f3fbf696 | 55967f63 |

| j = 10 |

| f477625b | 15bab3e6 | 5bfa627a | 945b9755 | d2b3c82b | 85bdad34 | c7c03fff | 4990985e |
| ecbfb29 | f477625b | 7567c2c2 | 5bfa627a | 604bda38 | 1a42df96 | c7c03fff | 4990985e |
| b9f6943d | ecbfb29 | 7567c2c2 | ecbfb29 | 604bda38 | 1a42df96 | 9e9a42d9 | c7c03fff |
| c537ac67 | b9f6943d | 7567c2c2 | 604bda38 | 1a42df96 | 9e9a42d9 | c7c03fff | 4990985e |
| c59665b7 | c537ac67 | ed287b73 | 7f47539d | ecbfb29 | 604bda38 | 1a42df96 | c7c03fff |
| 50115e1f | c59665b7 | ed287b73 | 7f47539d | ecbfb29 | 604bda38 | 1a42df96 | c7c03fff |
| 44196085 | 50115e1f | cbcc7b80 | 6f58cf8a | 0a2cc22a | 3d03e81b | 9e9a42d9 | c7c03fff |
| bde4e355 | 44196085 | cbcc7b80 | 6f58cf8a | 0a2cc22a | 3d03e81b | 9e9a42d9 | c7c03fff |
| ca176dca | bde4e355 | 32c10a88 | 22bc3ea0 | 0b418ac1b | 9e9a42d9 | c7c03fff | 4990985e |
| 541e456e | ca176dca | 32c10a88 | 32c10a88 | 32c10a88 | 32c10a88 | 32c10a88 | 32c10a88 |

| j = 20 |

| b6feeef7 | 541e456e | 2ed9b959 | c9c6ab7b | d41f5fda | 35cf8215 | 60d9a0c5 | 0b418ac1b |
| 0266e2f7 | b6feeef7 | c9c6ab7b | d41f5fda | 35cf8215 | 60d9a0c5 | 0b418ac1b | 9e9a42d9 |
| 8fd27582 | 0266e2f7 | fdddefd6 | 3c8adca8 | 84dc4c2 | c9365b1 | fed6a0f4 | 9e9a42d9 |
| 2527f8c6 | 8fd27582 | dc85e004 | fdddefd6 | 29d9c94d | a48d4c2 | 588e4a1b | fed6a0f4 |
| 3218579f | 2527f8c6 | a4eb051f | dc85e004 | 0da81ad7 | 29d9c94d | 2615246 | 588e4a1b |
| 35421cf3 | 3218579f | 4ff18c4a | a4eb051f | 644b37e4 | 0da81ad7 | 4ea594ee | 2615246 |
Appendix A.  Hash Value

debef9f9 2275b8a1 38604889 c18e5a4d 6fbd70e5 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 414C494345313233405941484F4F2E434F4D787968B4 FA32C3FD 2417842E 73BBFEFF 2F3C848B 6831D7E0 EC65228B 3937E49863E4C6D3 B23B0C84 9CF84241 484BFE48 F61D59A5 B16BA06E 6E12D1DA 27C5249A421DEBD6 1B62EAB6 746434EB C3CC315E 32220B3B ADD50BDC 4C4E6C14 7FEDD43D 0680512B CBB42C07 D47349D2 153B70C4 E5D7FDCC BFA36EA1 A85841B9 E46E09A20AE4C779 8AA0F119 471BEE11 825BE462 02BB79E2 A5844495 E97C04FF 4DF2548A7C0240F8 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 BFB7BC9A6D657373 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:
B.4. GB/T 32918.2-2016 A.3 Example 2

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

Input:

\begin{verbatim}
26352AF8 2EC19F20 7BBC6F94 74E11E90 CE0F7DDA CE03B27F 801817E8 97A81FD5
\end{verbatim}

Output:

\begin{verbatim}
AD673CBD A3114171 29A9EAA5 F9AB1AA1 633AD477 18A84DFD 46C17C6F A0AA3B12
\end{verbatim}

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:

\begin{verbatim}
0090 414C494345313233405941484F4F2E434F4D 00 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00 00E7BCD0 0090 41BC203 78A7E72B 12BCE002 66B9627E CB0B5A25 367AD1AD 4CC6242B 00 00CDB9CA 7E7968B4 FA32C3FD 2417842E 73BBFEFF 2F3C848B 6831D7E0 6C65228B 3937E498 6D684CE6D3 B23B0C84 9CF84241 484BFE48 F61D59A5 B16BA06E 6E12D120 AD27C5249A 421DEBD6 1B62EAB6 746434EB C3CC315E 32220B3B ADD50BDC 4C4E6C14 7FEDD43D 0680512B CBB42C07 D47349D2 15B370C4 E5D7FDFC BFA36EA1 A85841B9 E46E09A2 3099093B F3C137D8 FCBB0CF4 A22E50F3 B0F216C3 122D7942 5FE03A45 DBFE1655 3DF79E8D AC1CF0EC BAAF2F2B4 9D51A4B3 87F2EFAF 48233908 6A27A8E0 5BAED98B
\end{verbatim}

Output:
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 3F4C02A7
421DEB6E 1B62EAB6 74643E48 C3C3258E 32220B3B ADD50BDC 4C4E6C14 7FEDD43D
```

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, \[\text{Hash}(x_V \ | \ Z_A \ | \ Z_B \ | \ x_1 \ | \ y_1 \ | \ x_2 \ | \ y_2)\].

Input:

```
47C82653 4DC2F6F1 FB2F8426 DD658F21 E174F481 79ACEF29 00FB7F5 66E40905
E4D1D0C3 CA4C7F11 BC8FF8CB 3F4C02A7 8F108FA0 98E51A66 8487240F 75E20F31
6B4B6D0E 276691BD 4A11BF72 F4FB501A E309FDAC B72FA6CC 336E6656 119ABD67
6CB56338 16F4DD56 0B1DE4C5 8310CECC 6586C095 0532A46D 23150C40 8F162BF0
0D6FCF62 F1036C0A 1B6DACCF 57399223 A65F7D7B F2D9637E 5BBBEEB5 7961BFA1
179982A2 C7782953 00D9A232 5C686129 B8F2B533 7B3DCF45 14E8BBC1 9D900EE5
54C9288C 82733EFD F7808AE7 F27D0E73 2F7C73A7 D9AC98B7 D8740A91 D0DB3CF4
```

Output:

```
FF49D95B D45FCE99 ED54A8AD 7A709110 9F513944 429116BD 54D1DE43 79D97647
```

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

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

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 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 C95A0EEA 4BBD00CA 50B93EC2 4ACB6833 5D20BA5D CFE3B33B DBD2B62D 00

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 9F8DEACA56 21A99ACDF 2B557BAD 30E18355 9AEEC3B2 256E1C7C 11F870D2 2B165D01 5ACF9465 B09B87B5 27018107 6543ED19 058C38B3 13D73992 1D46B800 94D961A1 3673D4A5 CF8C7159 E30401D8 CFF7CA2 7A01A2E8 8C186737 48FDE9A7 4C1F9B45 64ECA09 97293C15 C34DE800 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 3748A372 D91610B6 98252CC9 E05FE287 B73B0CE6 639524CD 86694311 562914F4 F6A34241 01D885F8 8B05369C

Output:
4EB47D28 AD3906D6 244D010E 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 3748A372 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 9FB0D46D 20E9FFD5 B9F0DAB8
B.17. GB/T 32918.4-2016 A.3 Example 2

From [GB/T.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];
}

double_print("IV:
");
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;

double_print("\n== SM3 Message Expansion ME (sm3_me):
");
/* 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];
}

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

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

static void sm3_ce(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;

dump_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];

dump_print("A B C D E F G H\n");
dump_print("---------- initial value ----------\n");
dump_printf(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 + W[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 + W[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);
/* Update Context */
ctx->state[0] ^= A;
ctx->state[1] ^= B;

/* 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;
}

/*
designate("\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):

/* 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);

dbg_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...
");
/* 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.
");
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.
");
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 "+
"+++++++++++++++++++++++");
print_hash((unsigned*)ctx.state);

memcpy(digest, ctx.state, sizeof(uint32_t) * 8);

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.

#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);

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

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

int main(int argc, char **argv)
{
    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 +
    abcdeabcdeabcdeabcdeabcdeabcdeabcdeabcdeabcde" */
    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,
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,
    0x484BFE48, 0xF61D59A5, 0xB16BA06E, 0x6E12D1DA,
    0x27C5249A, 0x421DEBD6, 0x1B62EAB6, 0x746434EB,
    0xC3CC315E, 0x32220B3B, 0xAADD50BDC, 0x4C4E6C14,
    0x7FEDD43D, 0x0680512B, 0xCBB42C07, 0xD47349D2,
    0x153B70C4, 0xE5D7FDFC, 0x0BFA36EA1, 0xA85841B9,
    0x4E46E09A2, 0x0AAE4C779, 0x8AA0F119, 0x471BEE11,
    0x825BE462, 0x02BB79E2, 0xA584495, 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] = {
    0x4A38489, 0xE32B45B6, 0xF876E3AC, 0x2168CA39,
    0x2362DC8F, 0x23459C1D, 0x1146FC3D, 0xBFB7BC9A,
    0x6D573737, 0x61676520, 0x64696765, 0x00007473
}


```c
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, 0xF5F7777A, 0xe61181C38, 0xBAAE260B1,
    0x75951D8C, 0x060C2B3E, 0x01659616, 0x45281A86,
    0x26607B91, 0x7F657D7E, 0x9382F1EA, 0x5CD931F4,
    0x0F6627F3, 0x57542653, 0x8B2016E65, 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] = {
```
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, 0x6B16B06E, 0x6E12D1DA,
    0x27C5249A, 0x421DEBD6, 0x1B62EAB6, 0x746434EB,
    0xC3CC315E, 0x32220B3B, 0xADD50BDC, 0x4C4E6C14,
    0xFEDD43D, 0x0680512B, 0xCBB42C07, 0xD4734D92,
    0x15B70C4, 0xE5D7FDFC, 0xBFA36EA1, 0xA5841B9,
    0xE46E09A2, 0x3099093B, 0xF3C137D8, 0xFCBCCDF4,
    0xA2AE50F3, 0xB0F216C3, 0x122D7942, 0x5F03A45,
    0xDBFE1655, 0x3DF79E8D, 0xAC1CF0EC, 0xBAA2F2B4,
    0x9D51A4B3, 0x87F2EFAD, 0x48233908, 0x62A7A8E0,
    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 *

```c
static const uint32_t gbt329183m02[53] = {
    0x00884249, 0x4C4C3435, 0x36405941, 0x484F4F2E,
    0x434F4D78, 0x796B64FA, 0x32C3FD24, 0x17842E73,
    0xBBFEEFF2F, 0x3C848B68, 0x31D7E0EC, 0x65228B39,
    0x37E49863, 0xE4C6D3B2, 0x3B0C849C, 0xF8424148,
    0x4BE48F6, 0x1D59A5B1, 0x6BA06E6E, 0x12D1DA27,
    0x5C249A42, 0x1DEBD61B, 0x62EAB674, 0x6434EBC3,
    0xCC315E32, 0x220B3BAD, 0xD50BDC4C, 0x6E6C147F,
    0xEDD43D06, 0x80512BCB, 0xB42C07D4, 0x7349D215,
    0x3B70C4E5, 0xD7FDFCBF, 0xA36EA1A8, 0x5841B9E4,
    0x6E09A22A, 0x5493D446, 0xC38D8CC0, 0x119ABD67,
    0x90E7DF63, 0x3A8A4BF8, 0x3329B5EC, 0xE604B2B4,
    0xF37F4353, 0xC0B69F4B, 0x9E17773D, 0xE68FEC45,
    0xE14904E0, 0xDEA45BF6, 0xC85EA047, 0x004C0AC6
};
static const uint32_t gbt329183e02[8] = {
    0x6B4B6D0E, 0x276691BD, 0x4A11BF72, 0xF4FB501A,
    0xE309FDAC, 0xB72FA6CC, 0x0B72FA6CC, 0x336E6656,
};
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 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, 0x6BB5C68, 0x46A4C4A2,
    0x95FF49D9, 0x5BD45FCE, 0x99ED54A8, 0xAD7A7091,
    0x109F5139, 0x4442916B, 0xD154D1DE, 0x4379D976,
    0x00000047
};
static const uint32_t gbt329183e04[8] = {
    0x284C8F19, 0x8F141B50, 0x2E81250F, 0x1581C7E9,
    0xEBEB4CA9, 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, 0x6BB5C68, 0x46A4C4A2,
    0x95FF49D9, 0x5BD45FCE, 0x99ED54A8, 0xAD7A7091,
    0x109F5139, 0x4442916B, 0xD154D1DE, 0x4379D976,
    0x00000047
};
static const uint32_t gbt329183e05[8] = {
    0x23444DAF, 0x8ED75343, 0x66CB901C, 0x84B3BDDB,
    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, 0x00E78BCD, 0x09746C20, 0x2378A7E7,
    0x2B12BCE0, 0x0266B962, 0x7ECB0B5A, 0x25367AD1,
    0xAD4CC624, 0x2B00CDB9, 0xCA7F16E6B, 0x0441F658,
    0x343F4B10, 0x297C0EF9, 0xB6491082, 0x40A62E7,
    0xA7485735, 0xFADD013D, 0xE74DA659, 0x51C4D76D,
    0xC89220D5, 0xF7777A61, 0xB1C38BA, 0x260B175,
    0x951DC806, 0x8C2B3E00, 0x34297DD8, 0x3AB14D5B,
    0x393B6712, 0xF32B2F2E, 0x938D4690, 0xB095424B,
    0x89DA880C, 0x52D4A7D9, 0x0199BBF1, 0x1AC95A0E,
    0xA34BBD00, 0xCA50B93E, 0xC24ACB68, 0x335D20BA,
    0x5DCF3E3B3, 0x3BDE2B6, 0x0000000D
};
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,
    0x54ECEF008, 0x0215977B, 0x2E5D6D61, 0xB58A9944,
static const uint32_t gbt329183m08[8] = {
  0x2F03E880, 0x3DC39E34, 0x9F8DCA56, 0x21A9ACDF,
  0x2B557BAD, 0x30E18355, 0x9AEEC3B2, 0x256E1C7C,
  0x11F870D2, 0x2B165D01, 0x5ACF9465, 0xB09B87B5,
  0x7018107, 0x6543ED19, 0x058C38B3, 0x13D73992,
  0x1D46B800, 0x94D961A1, 0x3673D4A5, 0xCF8C7159,
  0xE30401D8, 0xCFFF7CA2, 0x7A01A2E8, 0x8C186737,
  0x48FDE9A7, 0x4C1F9B45, 0x6543ED19, 0x058C38B3,
  0x13D73992, 0x6543ED19, 0x058C38B3, 0x13D73992,
};
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, 0x2E87B73B, 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, 0x25B57996, 0x252CE5EB,
    0x086902B8C, 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,
0xDF24963F, 0x5DC9C648, 0x0566C80F, 0x8A1D8CC5,
0x0083E628, 0xCF701EE3, 0x141E8873, 0xFE55936A,
0x0078BF77
};
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] = {
0x083E628, 0xCF701EE3, 0x141E8873, 0xFE55936A,
0xFD24963F, 0x5DC9C648, 0x0566C80F, 0x8A1D8CC5,
0x1B656E63, 0x72797074, 0x696F6E20, 0x7374616E,
0x64617264, 0x01524C64, 0x7F0C0412, 0xDEFD468B,
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"
```c
#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

"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