

1 3GPP2 S.S0078-B

2 Version 1.0

3 Version Date: February 2008



3RD GENERATION
PARTNERSHIP
PROJECT 2
"3GPP2"

4
5
6
7
8
9
10 *Common Security Algorithms*

11
12
13
14
15
16
17
18
19
20
21
22
23
24 ***COPYRIGHT NOTICE***

3GPP2 and its Organizational Partners claim copyright in this document and individual Organizational Partners may copyright and issue documents or standards publications in individual Organizational Partner's name based on this document. Requests for reproduction of this document should be directed to the 3GPP2 Secretariat at secretariat@3gpp2.org. Requests to reproduce individual Organizational Partner's documents should be directed to that Organizational Partner. See www.3gpp2.org for more information.

25

1 **EDITOR**

2 *Frank Quick*
3 *Qualcomm Incorporated*
4 *5775 Morehouse Drive*
5 *San Diego, CA 92121 USA*
6 *fquick@qualcomm.com*

7 **REVISION HISTORY**

8

REVISION HISTORY		
Revision number	Content changes.	Date
1.0	<i>Initial Publication</i>	<i>12 06 2007</i>

Table of Contents

1

2	1. INTRODUCTION	1
3	1.1. Notations	1
4	1.2. Definitions	1
5	1.3. References	2
6	1.3.1. Normative	2
7	1.3.2. Informative	2
8	2. PROCEDURES	3
9	2.1. Hash Algorithms	3
10	2.1.1. SHA-1 and SHA-256	3
11	2.1.2. SHA-based MAC	4
12	2.1.2.1. SHA-1 Based MAC Calculation Procedure	4
13	2.1.2.2. SHA-256 Based MAC Calculation Procedure	6
14	2.1.2.3. UIM-Present MAC (UMAC) Generation Procedure	8
15	2.2. Authentication	9
16	2.2.1. UIM Authentication	9
17	2.2.2. One-Way Roaming to 2G systems	10
18	2.2.2.1. GSM Triplet Generation from SSD	10
19	2.2.2.2. 2G Key Generation from 3G Keys	12
20	2.3. Voice and Data Privacy	13
21	2.3.1. Encryption Key Generation	13
22	2.3.2. Key Strength Reduction	13
23	2.3.3. Enhanced Privacy Algorithm	14
24	2.3.3.1. Algorithm	14
25	2.3.3.2. ESP_privacykey Procedure	14
26	2.3.3.3. ESP_maskbits Procedure	15
27	2.3.3.4. ESP_AES Procedure	17
28	2.4. General Purpose Functions	17
29	2.4.1. SHA-Based Functions for f0 and f3	17
30	2.4.1.1. Constants	17
31	2.4.1.2. Random Number (RAND) Generation Procedure f0	18
32	2.4.1.3. Key Generation Procedure f3	21
33	3. REFERENCE IMPLEMENTATIONS	23
34	3.1. Privacy	23
35	3.1.1. Rijndael	23
36	3.1.2. Privacy Procedures	30
37	3.1.3. KeyStrengthRedAlg Function	33
38	3.2. Authentication	34
39	3.2.1. SHA-1	34
40	3.2.2. SHA-256	39

1	3.2.3.	Functions f0 and f3	44
2	3.2.4.	GSM Triplet Generation Function fh	48
3	3.2.5.	CDMA_3G_2G_Conversion Function	51
4	3.3.	EHMAC-SHA-1	52
5	3.4.	EHMAC-SHA-256	55
6	4.	TEST VECTORS	59
7	4.1.	Privacy	59
8	4.1.1.	Test Program Output	59
9	4.1.2.	Test Program	59
10	4.2.	Test Vectors for EHMAC-SHA-1	61
11	4.2.1.	Test Program Output	61
12	4.2.2.	Test Program	61
13	4.3.	Test Vectors for EHMAC-SHA-256	63
14	4.3.1.	Test Program Output	63
15	4.3.2.	Test Program	64
16	4.4.	Test Vectors for One-Way Roaming to 2G Systems	66
17	4.4.1.	Test Program Output	66
18	4.4.2.	Test Program	66
19	4.5.	Functions f0 and f3	68
20	4.5.1.	Test Program Output	68
21	4.5.2.	Test Program	69

List of Exhibits

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

EXHIBIT 2-1. PSEUDO RANDOM GENERATOR.....	20
EXHIBIT 2-2. KEY SCHEDULER.....	22
EXHIBIT 3-1 HEADER FOR RIJNDAEL.....	23
EXHIBIT 3-2 RIJNDAEL BOX DATA.....	23
EXHIBIT 3-3 RIJNDAEL ALGORITHM.....	25
EXHIBIT 3-4 HEADER FOR ESP.....	30
EXHIBIT 3-5 ESP_KEYSCHEM AND ESP_MASKBITS.....	31
EXHIBIT 3-6 KEYSTRENGTHREDALG FUNCTION HEADER.....	33
EXHIBIT 3-7 KEYSTRENGTHREDALG FUNCTION CODE.....	33
EXHIBIT 3-8 SHA-1 HEADER.....	34
EXHIBIT 3-9 SHA-1 CODE.....	34
EXHIBIT 3-10 SHA-256 CODE.....	39
EXHIBIT 3-11 F0F3 FUNCTION HEADER.....	44
EXHIBIT 3-12 F0F3 FUNCTION CODE.....	45
EXHIBIT 3-13 FUNCTION FH HEADER.....	48
EXHIBIT 3-14 FUNCTION FH CODE.....	48
EXHIBIT 3-15 CDMA_3G_2G_CONVERSION FUNCTION HEADER.....	51
EXHIBIT 3-16 CDMA_3G_2G_CONVERSION FUNCTION CODE.....	51
EXHIBIT 3-17 EHMALC HEADER.....	52
EXHIBIT 3-18 EHMALC CODE.....	52
EXHIBIT 3-19 UMAC_GENERATION CODE.....	55
EXHIBIT 3-20 EHMALC-SHA-256 HEADER.....	55
EXHIBIT 3-21 EHMALC-SHA-256 CODE.....	56
EXHIBIT 4-1 ESP_MASKBITS TEST OUTPUT.....	59
EXHIBIT 4-2 ESP_MASKBITS TEST PROGRAM.....	59
EXHIBIT 4-3 FUNCTIONS F0 AND F3 TEST OUTPUT.....	68
EXHIBIT 4-4 FUNCTIONS F0 AND F3 TEST PROGRAM.....	69

1. Introduction

1
2
3
4
5
6
7
8
9
10

This document defines detailed cryptographic procedures for common security algorithms in 3GPP2. The procedures include authentication algorithms and privacy algorithms that are intended to satisfy the export restriction requirements of 3GPP2 Organizational Partners' host countries. This document contains both textual descriptions and reference implementations for the procedures. The textual descriptions are provided as an aid to the reader. In the event of a conflict between the text description and the reference code, it is recommended that implementations agree with the reference code.

1.1. Notations

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

The notation 0x indicates a hexadecimal (base 16) number.

Binary numbers are expressed as a string of zero(s) and/or one(s) followed by a lower-case "b".

Data arrays are indicated by square brackets, as Array[]. Array indices start at zero (0). Where an array is loaded using a quantity that spans several array elements, the most significant bits of the quantity are loaded into the element having the lowest index. Similarly, where a quantity is loaded from several array elements, the element having the lowest index provides the most significant bits of the quantity.

Big-endian byte ordering is assumed in this specification.

This document uses ANSI C language programming syntax to specify the behavior of the cryptographic algorithms (see [2]). This specification is not meant to constrain implementations. Any implementation that demonstrates the same behavior at the external interface as the algorithm specified herein, by definition, complies with this standard.

1.2. Definitions

28
29
30
31
32
33
34

Internal Stored Data	Stored data that is defined locally within the cryptographic procedures and is not accessible for examination or use outside those procedures.
MSB	Most Significant Bit.
XOR	Bitwise logical exclusive or function.
Word	A data unit that contains 32 bits or 4 bytes where byte 0 is the most significant byte and byte 3 is the least significant byte.

1.3. References

1.3.1. Normative

1. Federal Information Processing Standard FIPS 180-2, “Secure Hash Standard,” August 1, 2002

1.3.2. Informative

2. ANSI/ISO 9899-1999, “Programming Languages - C”
3. A Million Random Digits with 100,000 Normal Deviates, The RAND Corporation, 1955, online at <http://www.rand.org/publications/classics/randomdigits> .
4. Federal Information Processing Standard FIPS 197, “Advanced Encryption Standard (AES),” November 26, 2001.
5. S.S0053, “Common Cryptographic Algorithms”.
6. N.S0005, “Cellular Radiotelecommunications Intersystem Operations”.

2. Procedures

2.1. Hash Algorithms

2.1.1. SHA-1 and SHA-256

The hash functions used in this document are SHA-1 and SHA-256, defined in [1]. Refer to 3.2.1 for a reference implementation of the SHA-1 algorithm. In this document, the function $F()$ refers to the SHA-1 or SHA-256 algorithm.

Test vectors for SHA-1 and SHA-256 are given in [1].

SHA-1 and SHA-256 use an iterated construction where the input message is processed block by block. The basic building block is called the compression function. The compression function used in this document differs from the SHA-1 and SHA-256 hash functions defined in [1] by the way its payload and chaining variable inputs are loaded. In this document, the function $f_K()$ refers to the compression function with key K exclusive-ored with the initialization vector.

2.1.2. SHA-based MAC

2.1.2.1. SHA-1 Based MAC Calculation Procedure

Procedure name:	ehmacsha	
Inputs from calling process:		
key_length	integer	
key	8*key_length bits	
message	message_length bits	
message_length	integer	
message_offset	integer	
MAC_length	integer	
Inputs from internal stored data:	None.	
Outputs to calling process:		
MAC	8*MAC_length bits	
Outputs to internal stored data:	None.	

The ehmacsha procedure computes a message authentication code (MAC) using a secret key. Refer to 3.3 for a reference implementation of the ehmacsha algorithm.

The MAC initialization procedures for the MAC calculation should be performed whenever a new key is generated. Initialization shall proceed as follows:

1. Define two strings: ipad = the byte 0x36 repeated 64 times and opad = the byte 0x5C repeated 64 times.
2. append zeros to the end of the key to create a 64 byte string.
3. XOR (bitwise exclusive-OR) the 64 byte string computed in step 2 with ipad defined in step 1.
4. Apply SHA-1 compression function (see 2.1.1) to the 64-byte data string computed in step 3 loaded in the function payload, while chaining variable input is loaded with the standard SHA-1 initial hash value defined in [1].

- 1 5. Store the result of SHA-1 compression conducted in step 4 as
2 the intermediary key K1.
- 3 6. XOR (bitwise exclusive-OR) the 64 byte string computed in
4 step 2 with opad defined in step 1.
- 5 7. Apply SHA-1 compression function (see 2.1.1) to the 64-byte
6 data string computed in step 6 loaded in the function payload,
7 while chaining variable input is loaded with the standard
8 SHA-1 initial hash value defined in [1].
- 9 8. Store the result of SHA-1 compression conducted in step 7 as
10 the intermediary key K2

11 Computation of MAC should proceed as follows: If length of the
12 Message is > 510 bits,

- 13 1. Split the message into the Suffix, M_{suff} , as the least significant
14 part of the message with the length of 351 bits, and the Prefix,
15 M_{pref} , as the remaining most significant part of the message
16 with the Length = (length of message – 351 bits).
- 17 2. Apply the SHA-1 compression function (see 2.1.1) with the
18 key K1 loaded in the chaining variable input to the padded
19 M_{pref} to compute $Y = f_{K1}(M_{\text{pref}}, \text{pad}, \text{Length})$.
- 20 3. Assemble the bit string containing the keyed hash digest Y of
21 the Prefix , followed by the message Suffix , followed by an
22 Indicator bit set to '1'.
- 23 4. Apply the SHA-1 compression function (see 2.1.1) with the
24 key K2 loaded into the chaining variable input to the bit string
25 assembled in step 3 to compute the MAC of the message:
26 $\text{MAC} = f_{K2}(Y, M_{\text{suff}}, 1)$.

27 If length of the Message is <= 510 bits,

- 28 1. Append the padding bit set to '1' to the least significant end of
29 the message.
- 30 2. Append as many '0' bits as necessary to extend the length of
31 resulting bit string to 511 bits.
- 32 3. Append the least significant bit of the resulting bit string with
33 an Indicator bit set to '0'.
- 34 4. Apply the SHA-1 compression function (see 2.1.1) loaded
35 with the key K2 at the chaining variable input to the bit string
36 assembled in step 3 to compute the MAC of the message:
37 $\text{MAC} = f_{K2}(M, \text{pad}, 0)$.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35

Test vectors for the MAC procedure can be found in 4.2.

2.1.2.2. SHA-256 Based MAC Calculation Procedure

Procedure name:	
ehmacsha256	
Inputs from calling process:	
key_length	integer
key	8*key_length bits
message	message_length bits
message_length	integer
message_offset	integer
MAC_length	integer
Inputs from internal stored data:	
None.	
Outputs to calling process:	
MAC	8*MAC_length bits
Outputs to internal stored data:	
None.	

The ehmacsha256 procedure computes a message authentication code (MAC) using a secret key.

The MAC initialization procedures for the MAC calculation should be performed whenever a new key is generated. Initialization shall proceed as follows:

1. Define two strings: ipad = the byte 0x36 repeated 64 times and opad = the byte 0x5C repeated 64 times.
2. append zeros to the end of the key to create a 64 byte string.
3. XOR (bitwise exclusive-OR) the 64 byte string computed in step 2 with ipad defined in step 1.
4. Apply SHA-256 compression function to the 64-byte data string computed in step 3 loaded in the function payload, while chaining variable input is loaded with the standard SHA-256 initial hash value defined in [1].
5. Store the result of SHA-256 compression conducted in step 4 as the intermediary key K1.

- 1
2
6. XOR (bitwise exclusive-OR) the 64 byte string computed in step 2 with opad defined in step 1.
- 3
4
5
6
7. Apply SHA-256 compression function (see 2.1.1) to the 64-byte data string computed in step 6 loaded in the function payload, while chaining variable input is loaded with the standard SHA-256 initial hash value defined in [1].
- 7
8
8. Store the result of SHA-256 compression conducted in step 7 as the intermediary key K2

9
10

Computation of MAC should proceed as follows: If length of the Message is > 510 bits,

- 11
12
13
14
1. Split the message into the Suffix, M_{suffix} , as the least significant part of the message with the length of 255 bits, and the Prefix, M_{prefix} , as the remaining most significant part of the message with the Length = (length of message – 255 bits).
- 15
16
17
2. Apply the SHA-256 compression function (see 2.1.1) with the key K1 loaded in the chaining variable input to the padded M_{prefix} to compute $Y = f_{K1}(M_{\text{prefix}}, \text{pad}, \text{Length})$.
- 18
19
20
3. Assemble the bit string containing the keyed hash digest Y of the Prefix, followed by the message Suffix, followed by an Indicator bit set to '1'.
- 21
22
23
24
4. Apply the SHA-256 compression function (see 2.1.1) with the key K2 loaded into the chaining variable input to the bit string assembled in step 3 to compute the MAC of the message: $\text{MAC} = f_{K2}(Y, M_{\text{suffix}}, 1)$.

25

If length of the Message is ≤ 510 bits,

- 26
27
1. Append the padding bit set to '1' to the least significant end of the message.
- 28
29
2. Append as many '0' bits as necessary to extend the length of resulting bit string to 511 bits.
- 30
31
3. Append the least significant bit of the resulting bit string with an Indicator bit set to '0'.
- 32
33
34
35
4. Apply the SHA-256 compression function (see 2.1.1) loaded with the key K2 at the chaining variable input to the bit string assembled in step 3 to compute the MAC of the message: $\text{MAC} = f_{K2}(M, \text{pad}, 0)$.

36

2.1.2.3. UIM-Present MAC (UMAC) Generation Procedure

1	Procedure name:	
2	UMAC_Generation	
3	Inputs from calling process:	
4	UAK	8*UAK_length bits
5	UAK_length	integer
6	MAC_length	integer
7	MAC	8*MAC_length bits
8	Inputs from internal stored data:	
9	None.	
10	Outputs to calling process:	
11	UMAC	8*MAC_length bits
12	Outputs to internal stored data:	
13	None.	
14		
15		
16		

The function UMAC_Generation is used to compute UMAC, which is a hash of a MAC using UAK. Since UMAC can only be computed within the UIM, it provides a means for the mobile station to prove that the UIM was present at the time the message is formed.

The function *fill* must be called prior to calling UMAC_Generation.

The UMAC_Generation function computes the ehmacsha hash (see 2.1.2.1) of MAC, treated as a message of length 8*MAC_length. UMAC_Generation returns the most-significant bits of the message digest as UMAC, having the same size as the MAC.

The UMAC is generated in two steps, as follows:

The first step is to calculate a MAC of the message (for example, as described in 2.1.2.1). This step can be performed in the ME shell.

The second part of the UMAC calculation uses the procedure UMAC_Generation to perform a keyed hash of the result of the first

1 part using SHA-1 based MAC. The UMAC_Generation procedure
2 shall be performed in the UIM.¹.

3 The UMAC_Generation procedure proceeds as follows:

- 4 1. Assemble the bit string containing the MAC of the message as
5 described in 2.1.2.1.
- 6 2. Append the padding bit set to '1' to the least significant end of
7 the message.
- 8 3. Append as many '0' bits as necessary to extend the length of
9 resulting bit string to 511 bits. Append the least significant bit
10 of the resulting bit string with an Indicator bit set to '0'.
- 11 4. Apply the SHA-1 compression function (see 2.1.1) loaded
12 with the key UAK XOR (bitwise exclusive-OR) with the
13 standard SHA-1 initial hash value defined in [1] at the
14 chaining variable input to the bit string assembled in step 3 to
15 compute the UMAC of the message: $UMAC =$
16 $f_{UAK \oplus IV}(MAC, pad, 0)$.

17 A test vector for UMAC_Generation is included in 4.2.

18 2.2. Authentication

19 2.2.1. UIM Authentication

20 The function in this section is an extension to AKA that can be used in
21 local authentication procedures. These procedures are used to prove the
22 presence of the UIM during cellular operations.

23 Procedure *UMAC_Generation* (see 2.1.2.3) computes the SHA-1 hash
24 of a MAC (typically keyed with IK), keyed with UAK.

¹ UAK is typically used only in a removable UIM. It is possible to use UAK with a non-removable UIM, but there is no security reason for doing so.

2.2.2. One-Way Roaming to 2G systems

2.2.2.1. GSM Triplet Generation from SSD

3	Procedure name:	
4	fh	
5	Inputs from calling process:	
6	Shared Secret Data (SSD):	
7	SSD_A	64 bits
8	SSD_B	64 bits
9	Random Number (RAND)	128 bits
10	Type identifier (fh)	8 bits
11	Family Key (Fmk)	32 bits
12	Inputs from internal stored data:	
13	None.	
14	Outputs to calling process:	
15	GSM Triplet:	
16	RAND	128 bits
17	SRES	32 bits
18	Kc	64 bits
19	Outputs to internal stored data:	
20	None.	
21		

The function fh can be used to generate a GSM authentication triplet to support one-way roaming from ANSI/TIA-41 [6] systems to GSM systems. On the network side, this function is typically performed in an Interoperability and Interworking Function (IIF), which obtains SSD from the home system as would an ANSI/TIA-41 VLR, and creates one or more GSM triplets that are sent to GSM visited systems.

The following constants are used in this procedure:

fh = 0x60

The family key Fmk can be set to any desired value, but the same value must be used in the mobile station and in the IIF. Unless otherwise specified, the value of Fmk should be set to 0x42454c4c.

Procedure:

1. Load the registers of SHA-1 with known constants as follows:
Load the initial hash value (IV) with the standard SHA-1 initial hash value.

- 1 Load the 512-bit payload (16 32-bit words) with the constant 0x5C
2 repeated 64 times.
- 3 2. Load the SSD parameters and an 64-bit internal counter as
4 follows: XOR the SSD-A into the leftmost (most significant) 64
5 bits of IV, and XOR the SSD-B into the next 64 bits of IV. The
6 64-bit internal counter, initialized to 0, is XORed into the (0th, 1st)
7 words, (4th, 5th) words, (8th, 9th) words, and (12th, 13th) words of
8 the payload. Next, a Type Identifier fh is XORed into the 2nd word,
9 and the family key is XORed with the 3rd word of the payload. The
10 128-bit random number is split into two parts (64 bits each). The
11 least significant 64 bits are XORed with the 6th and 7th words, and
12 the most significant 64 bits are XORed with the 10th and 11th
13 words of the payload.
- 14 3. Run SHA-1 to produce the 160-bit output.
- 15 4. The polynomial $AX + B \text{ mod } G$ is calculated, where: A and B are
16 predetermined 160-bit random numbers (treated as polynomials
17 with binary coefficients in the variable T). X is the 160-bit output
18 from the SHA-1 operation, treated as a polynomial with binary
19 coefficients in the variable T. G is the polynomial $T^{160} + T^5 + T^3$
20 $+ T^2 + 1$. Extract the least significant 64 bits and store it in the key
21 buffer.
- 22 5. Steps 2 through 5 are repeated 2 times, with the index incremented
23 between iterations. This gives a total of 128 bits in the key buffer.
- 24 6. Set the RAND parameter of the GSM triplet to the value of the
25 Random Number RAND.
- 26 7. Set the Kc parameter of the GSM triplet to the first 64 least
27 significant bits of the key buffer of Step 6.
- 28 8. Set the SRES parameter of the GSM triplet to the next 32 least
29 significant bits of the key buffer of Step 6.
- 30 See 3.2.3 for a description of this algorithm in ANSI C.

31

2.2.2.2. 2G Key Generation from 3G Keys

1	Procedure name:	
2	CDMA_3G_2G_Conversion	
3	Inputs from calling process:	
4	AKA Ciphering Key (CK)	128 bits
5	Inputs from internal stored data:	
6	None.	
7	Outputs to calling process:	
8	PLCM	40 bits
9	CMEAKEY	64 bits
10	Outputs to internal stored data:	
11	None.	
12		
13		

14 This procedure can be used to create the CDMA private long code
 15 mask (PLCM) and the message encryption key CMEAKEY for
 16 intersystem handoff from a system using AKA to a system using older
 17 (2G) algorithms for authentication and privacy. (On the ANSI/TIA-41
 18 network [6], these keys are referred to as CDMA_PLCM and
 19 SMEKEY.)

20 Note that this procedure does not create SSD-A, and therefore this
 21 procedure does not provide support for the Unique Challenge-
 22 Response procedure after an intersystem handoff to a 2G system.

23 Procedure:

- 24 1. Calculate the 160-bit SHA-1 digest of the string consisting of
 25 the 160 bits of the ASCII string
 26 "3G_2GCDMA_CONVERSION"
 27 concatenated with the 128 bits of the ciphering key CK.
- 28 2. Set the Private Long Code Mask PLCM to the 40 least
 29 significant bits of the SHA-1 output.
- 30 3. Set the CMEA Key CMEAKEY to the next 64 least
 31 significant bits of the SHA-1 output.

32 See 3.2.5 for a description of this algorithm in ANSI C.

33

2.3. Voice and Data Privacy

2.3.1. Encryption Key Generation

When the Mobile Station performs authentication in accordance with air interface procedures that invoke the CAVE algorithm (see [5]), the key used for the Rijndael encryption algorithm should be the 64-bit CMEAKEKEY (also sent on the ANSI/TIA-41 network [6] as SMEKey), repeated twice to form the 128-bit key that is input to the ESP_privacykey procedure.

2.3.2. Key Strength Reduction

Procedure name:

KeyStrengthRedAlg

Inputs from calling process:

KeyLength	integer
OriginalKey	KeyLength octets
SaltLength	integer
Salt	SaltLength octets
KeyEntropy	integer, 0..16

Inputs from internal stored data:

None

Outputs to calling process:

RedStrengthKey	KeyLength octets
----------------	------------------

Outputs to internal stored data:

None.

Here are the steps to create the reduced-strength key:

1. The hash function is applied to the concatenation of OriginalKey and Salt, to form an intermediate key K'. That is,

$$K' = \text{SHA}(\text{OriginalKey} \parallel \text{Salt});$$

2. K' is modified so that it contains only KeyEntropy meaningful octets, by setting the leftmost (20-KeyEntropy) octets to zero.

3. SHA-1 is applied to the key K' concatenated with Salt to form an output value KP. That is,

$$KP = \text{SHA}(K' \parallel \text{Salt}).$$

1 4. The leftmost KeyLength octets of KP are output as
2 RedStrengthKey.

3 See 3.1.3 for a description of this algorithm in ANSI C.

4

5 **2.3.3. Enhanced Privacy Algorithm**

6 **2.3.3.1. Algorithm**

7 The enhanced privacy algorithm uses 128-bit Rijndael (see [4]). Refer
8 to 3.1 for a reference implementation of the enhanced privacy
9 procedures using Rijndael. Test vectors for the enhanced privacy
10 algorithm are provided in 4.1.

11 **2.3.3.2. ESP_privacykey Procedure**

12	Procedure name:
13	ESP_privacykey
14	Inputs from calling process:
15	key 16*8 bits
16	Inputs from internal stored data:
17	None.
18	Outputs to calling process:
19	None.
20	Outputs to internal stored data:
21	ESP_privacykeyschedule
22	

23 This procedure accepts a 128-bit key, and uses it to initialize internal
24 data structures.

25 This procedure shall be performed prior to invoking the ESP_maskbits
26 procedure each time a new or different key comes into use.

27 Procedure:

28 Load the 128-bit ciphering key CK in a 4 by 8 array and invoke
29 rijndaelKeySched().

2.3.3.3. ESP_maskbits Procedure

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

Procedure name:	ESP_maskbits	
Inputs from calling process:		
	fresh	freshsize*8 bits
	freshsize	integer (1 .. 12)
	buf	pointer
	bit_offset	integer
	bit_count	integer
Inputs from internal stored data:	ESP_privacykeyschedule	
Outputs to calling process:		
	buf	xored with privacy mask
Outputs to internal stored data:	None.	

This procedure encrypts or decrypts data in *buf* by XORing into it a privacy mask of length *bit_count*, starting at the bit offset indicated by *bit_offset*. The octets in *buf* are assumed to be most-significant first, and the first bit of the buffer is the most significant bit of the first octet. Only the bits from *bit_offset* through $(bit_offset+bit_count-1)$ are changed. The mask bits are shifted to align with the bit offset, so that encryption and decryption buffers need not have the same bit offset. Since the underlying algorithm is essentially a 128-bit block cipher, to encrypt or decrypt data in *buf* that is larger than 128 bits, the cipher is executed more than once in a counter mode.

The inputs to the encryption process are:

- *freshsize*: size in octets of the variable *fresh*. *freshsize* shall be in the range 1 to 12, inclusive.
- *fresh*: *freshsize* octets provided directly by the calling process, to be used to vary the output on a per-buffer basis.
- *buf*: the address of a buffer to be encrypted or decrypted.
- *bit_offset*: the starting bit in the buffer to be encrypted or decrypted.
- *bit_count*: the number of bits of the buffer to be encrypted or decrypted.

1 Implementations using data encryption shall comply with the following
 2 requirements. These requirements apply to all data encrypted during a
 3 call.

- 4 • A privacy mask produced using a particular value of *fresh* should
 5 be used to encrypt only one set of data.
- 6 • A privacy mask produced using a value of *fresh* shall not be used
 7 to encrypt data in more than one direction of transmission.
- 8 • A privacy mask produced using a value of *fresh* shall not be used
 9 to encrypt data on more than one logical channel.

10 Procedure:

- 11 1. Initialize offset *bit_offset* to the starting bit in the buffer to be
 12 encrypted or decrypted.
- 13 2. Initialize pointer *buf* to the output buffer *buf* that will contain the
 14 encrypted or decrypted bits.
- 15 3. Initialize an internal 32-bit counter to zero.
- 16 4. Run the following until all data have been encrypted or decrypted:

17 Load buffer *b* (a 4 by 8 array) with 4 copies of the 32-bit internal
 18 counter, MSB first. Note that only the first half of the buffer is
 19 initialized and the other half is neither initialized nor used.

20 Overwrite the first *freshsize* bytes of buffer *b* with *fresh*.

21 Invoke `rijndaelEncrypt()` with *b*, `KEYLENGTH*8`,
 22 `BLOCKLENGTH*8`, and `ESP_privacykeyschedule`.

23 For the offset not starting at a byte boundary, set last mask octet to
 24 zero and set the first mask and its size.

25 For the offset starting at a byte boundary, set the first mask and its
 26 size.

27 XOR the mask into buffer

28 Check to see if there is any more data to be encrypted or
 29 decrypted.

30 Increment counter.

2.3.3.4. ESP_AES Procedure

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

Procedure name:	
ESP_AES	
Inputs from calling process:	
key	16*8 bits
fresh	freshsize*8 bits
freshsize	integer (1..12)
buf	pointer
bit_offset	integer
bit_count	integer
Inputs from internal stored data:	
None	
Outputs to calling process:	
buf	xored with privacy mask
Outputs to internal stored data:	
ESP_privacykeyschedule	

18
19
20
21
22
23
24

This procedure calls the ESP_privacykey procedure followed by the ESP_maskbits procedure, so that the complete ESP AES Rijndael algorithm can be invoked from a common interface.

Procedure:

1. Invoke ESP_privacykey() with ciphering key CK.
2. Invoke ESP_maskbits() with *fresh*, *freshsize*, *buf*, *bit_offset*, and *bit_count*.

25

2.4. General Purpose Functions

26

2.4.1. SHA-Based Functions for f0 and f3

27
28
29

This section provides the interface to a reference implementation of the functions f0 and f3 using the compression function of SHA-1. Refer to 3.2.3 for the reference implementation of f0 and f3.

30

2.4.1.1. Constants

31
32

The following constants are used in these functions: Fmk, A, B, f0, and f3. Constants A and B are 160 bits, constants f0 and f3 are 8 bits.

1 Fmk is a Family Key, should be defined with different values in
 2 different usages of these general-purpose functions for the purpose of
 3 cryptographic separation. .

4 A and B are as the least significant bits of the first 320 digits of the
 5 RAND Corporation book of random numbers [3]. Bit 1 and bit 161 are
 6 the MSB bits of A and B respectively. A and B are specified as
 7 follows:

8 A = 0x9DE9C9C8EFD5781148231401901F2D493F4C6365
 9 B = 0x75EFD15C4B8F8F514EF3BCC3794A765E7EEC45E0

10 The values of the 8-bit Type Identifiers for the functions f0 and f3 are
 11 specified as follows:

12 f0 = 0x41
 13 f3 = 0x45

14 **2.4.1.2. Random Number (RAND) Generation Procedure f0**

15	Procedure name:	
16	f0	
17	Inputs from calling process:	
18	Random Secret Seed (K)	128 bits
19	Type Identifier (f0)	8 bits
20	Family Key (Fmk)	32 bits
21	Inputs from internal stored data:	
22	Counter (counter)	64 bits
23	Outputs to calling process:	
24	RAND	64 bits
25	Outputs to internal stored data:	
26	None.	
27		

28 The function f0 can be used to generate pseudo random values (e.g.
 29 RAND values to be used in Authentication Vectors).

30 The function f0 is a pseudo random generator algorithm that is
 31 provably as hard as SHA-1. The function is presented with the random
 32 secret Seed, which is chosen by the operator, as well as a Counter
 33 parameter. The procedure returns 64 pseudo random bits every time it
 34 is invoked. The calling process is responsible for incrementing the
 35 counter and repeatedly calling the procedure in order to generate the
 36 required number of pseudo random bits.

1 When f0 is used to generate RAND values to be used in Authentication
2 Vectors, the Counter parameter is initialized to 0 at the Authentication
3 Center once, and is incremented every time the function is called.

4 When f0 is used for producing two or more keys from a common
5 secret, the calling process shall set the Counter value to zero for the
6 first invocation. If f0 is used for this purpose in the Authentication
7 Center, the Authentication Center is responsible for saving the value of
8 Counter used for RAND generation and restoring it before computing
9 additional values of RAND.

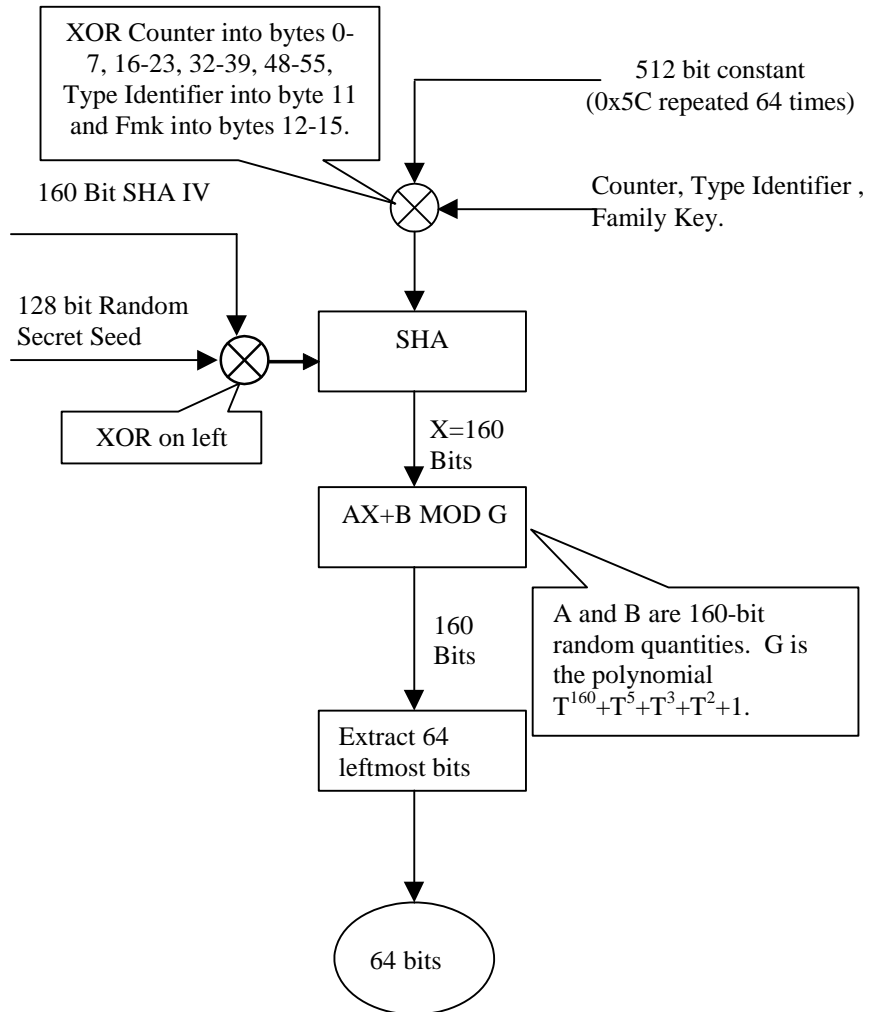
10 Procedure:

- 11 1. Load the registers of SHA-1 with known constants as follows:
12 Load the initial hash value (IV) with the standard SHA-1
13 initial hash value
14 Load the 512-bit payload with the constant 0x5C repeated 64
15 times
- 16 2. Load the Seed, Family Key, Type Identifier and Counter values as
17 follows:
18 XOR the Seed into the leftmost 128 bits of IV. The remaining
19 variables are XORed into payload in the following places with the
20 leftmost byte of the payload called the 0th byte. The 64-bit Counter
21 value is XORed into the 0th to 7th bytes, the 16th to 23rd bytes, the
22 32nd to 39th bytes and the 48th to 55th bytes. Next, the Type
23 Identifier (unique to function f0) is XORed into the 11th byte, and
24 the Family Key is XORed into the 12th to 15th bytes.
- 25 3. Run SHA-1 to produce a 160-bit output.
- 26 4. The polynomial $(AX + B \text{ mod } G)$ is calculated, where:
27 A and B are predetermined 160-bit random numbers (treated
28 as polynomials with binary coefficients in the variable T).
29 X is the 160-bit output from the SHA-1 operation, treated as a
30 polynomial with binary coefficients in the variable T.
31 G is the polynomial $T^{160} + T^5 + T^3 + T^2 + 1$.
- 32 5. The Counter is increased by 1, to ensure a different value is used
33 the next time the function is called. The leftmost 64 bits of the
34 result are returned and stored in a buffer.
- 35 6. Repeat procedure as many times as needed to obtain the required
36 number of bits. (When fewer than 64 bits are needed, the leftmost
37 bits should be used and the rest discarded.)

38 Steps 1-5 are illustrated in Exhibit 2-1.

1

Exhibit 2-1. Pseudo Random Generator.



2
3

2.4.1.3. Key Generation Procedure f3

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Procedure name:	
f3	
Inputs from calling process:	
Subscriber Authentication Key (K)	128 bits
Type Identifier (f3)	8 bits
Random Number (RAND)	128 bits
Family Key (Fmk)	32 bits
Inputs from internal stored data:	
None.	
Outputs to calling process:	
f3 Output Key (f3K)	128 bits
Outputs to internal stored data:	
None.	

16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

The function f3 is a pseudo random function used to generate a key. The output can be used as an encryption key, or for other purposes.

Procedure:

1. Initialize an 8 bit Counter to 0
2. Load the registers of SHA-1 with known constants as follows:
 - Load the initial hash value (IV) with the standard SHA-1 initial hash value
 - Load the 512-bit payload with the constant 0x5C repeated 64 times
3. Load the Subscriber Authentication Key, Type Identifier, Random Number, Family Key and Counter as follows:
 - XOR the subscriber authentication key into the leftmost 128 bits of IV. The remaining variables are XORed into payload in the following places with the leftmost byte of the payload called the 0th byte. The Counter is XORed into the 3rd, 19th, 35th and 51st bytes. Next, the Type Identifier (unique to function f3) is XORed into the 11th byte, and the family key is XORed into the 12th to 15th bytes. The 128-bit random number is XORed into the 24th to 39th bytes.
4. Run SHA-1 to produce the 160-bit output.
5. The polynomial $AX + B \text{ mod } G$ is calculated, where:
 - A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T).
 - X is the 160-bit output from the SHA-1 operation, treated as a polynomial with binary coefficients in the variable T.

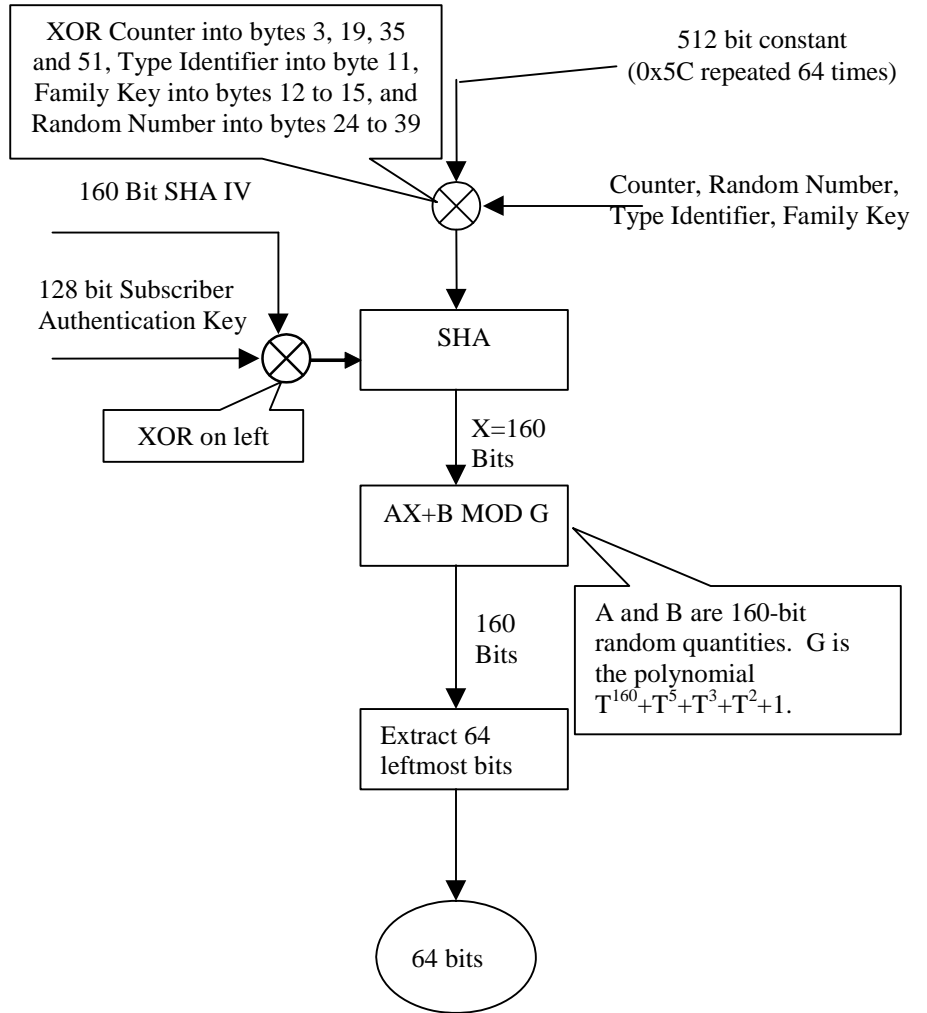
1
2
3
4
5
6
7

G is the polynomial $T^{160}+T^5+T^3+T^2+1$. Extract the leftmost 64 bits and store it in the key buffer.

6. The Counter is incremented by one and steps 2 through 5 are repeated one more time. This gives a total of 128 bits. Store these 128 bits in f3K.

Steps 2-5 are illustrated in Exhibit 2-2.

Exhibit 2-2. Key Scheduler.



8
9
10

3. Reference Implementations

3.1. Privacy

3.1.1. Rijndael

A reference implementation of the Rijndael algorithm is given below. Note that only the functions rijndaelKeySched and rijndaelEncrypt are used for CDMA enhanced privacy.

Exhibit 3-1 Header for Rijndael

```

8 /* rijndael-alg-ref.h v2.0 August '99
9  * Reference ANSI C code
10  * authors: Paulo Barreto
11  *          Vincent Rijmen
12  */
13 #ifndef __RIJNDAEL_ALG_H
14 #define __RIJNDAEL_ALG_H
15
16 #define MAXBC      (256/32)
17 #define MAXKC      (256/32)
18 #define MAXROUNDS  14
19
20 typedef unsigned char    word8;
21 typedef unsigned short   word16;
22 typedef unsigned long    word32;
23
24
25 int rijndaelKeySched (word8 k[4][MAXKC], int keyBits, int blockBits,
26                     word8 rk[MAXROUNDS+1][4][MAXBC]);
27 int rijndaelEncrypt (word8 a[4][MAXBC], int keyBits, int blockBits,
28                     word8 rk[MAXROUNDS+1][4][MAXBC]);
29 int rijndaelEncryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
30                           word8 rk[MAXROUNDS+1][4][MAXBC], int rounds);
31 int rijndaelDecrypt (word8 a[4][MAXBC], int keyBits, int blockBits,
32                     word8 rk[MAXROUNDS+1][4][MAXBC]);
33 int rijndaelDecryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
34                           word8 rk[MAXROUNDS+1][4][MAXBC], int rounds);
35
36 #endif /* __RIJNDAEL_ALG_H */

```

Exhibit 3-2 Rijndael Box Data

```

39 /* "boxes-ref.dat" */
40
41 word8 Logtable[256] = {
42  0,  0,  25,  1,  50,  2,  26, 198,  75, 199,  27, 104,  51, 238, 223,  3,
43 100,  4, 224, 14,  52, 141, 129, 239,  76, 113,  8, 200, 248, 105,  28, 193,
44 125, 194,  29, 181, 249, 185,  39, 106,  77, 228, 166, 114, 154, 201,  9, 120,
45 101,  47, 138,  5,  33,  15, 225,  36,  18, 240, 130,  69,  53, 147, 218, 142,
46 150, 143, 219, 189,  54, 208, 206, 148,  19,  92, 210, 241,  64,  70, 131,  56,
47 102, 221, 253,  48, 191,  6, 139,  98, 179,  37, 226, 152,  34, 136, 145,  16,
48 126, 110,  72, 195, 163, 182,  30,  66,  58, 107,  40,  84, 250, 133,  61, 186,
49  43, 121,  10,  21, 155, 159,  94, 202,  78, 212, 172, 229, 243, 115, 167,  87,
50 175,  88, 168,  80, 244, 234, 214, 116,  79, 174, 233, 213, 231, 230, 173, 232,
51  44, 215, 117, 122, 235,  22,  11, 245,  89, 203,  95, 176, 156, 169,  81, 160,
52 127,  12, 246, 111,  23, 196,  73, 236, 216,  67,  31,  45, 164, 118, 123, 183,
53 204, 187,  62,  90, 251,  96, 177, 134,  59,  82, 161, 108, 170,  85,  41, 157,

```

```

1 151, 178, 135, 144, 97, 190, 220, 252, 188, 149, 207, 205, 55, 63, 91, 209,
2 83, 57, 132, 60, 65, 162, 109, 71, 20, 42, 158, 93, 86, 242, 211, 171,
3 68, 17, 146, 217, 35, 32, 46, 137, 180, 124, 184, 38, 119, 153, 227, 165,
4 103, 74, 237, 222, 197, 49, 254, 24, 13, 99, 140, 128, 192, 247, 112, 7,
5 };
6
7 word8 Alogtable[256] = {
8 1, 3, 5, 15, 17, 51, 85, 255, 26, 46, 114, 150, 161, 248, 19, 53,
9 95, 225, 56, 72, 216, 115, 149, 164, 247, 2, 6, 10, 30, 34, 102, 170,
10 229, 52, 92, 228, 55, 89, 235, 38, 106, 190, 217, 112, 144, 171, 230, 49,
11 83, 245, 4, 12, 20, 60, 68, 204, 79, 209, 104, 184, 211, 110, 178, 205,
12 76, 212, 103, 169, 224, 59, 77, 215, 98, 166, 241, 8, 24, 40, 120, 136,
13 131, 158, 185, 208, 107, 189, 220, 127, 129, 152, 179, 206, 73, 219, 118, 154,
14 181, 196, 87, 249, 16, 48, 80, 240, 11, 29, 39, 105, 187, 214, 97, 163,
15 254, 25, 43, 125, 135, 146, 173, 236, 47, 113, 147, 174, 233, 32, 96, 160,
16 251, 22, 58, 78, 210, 109, 183, 194, 93, 231, 50, 86, 250, 21, 63, 65,
17 195, 94, 226, 61, 71, 201, 64, 192, 91, 237, 44, 116, 156, 191, 218, 117,
18 159, 186, 213, 100, 172, 239, 42, 126, 130, 157, 188, 223, 122, 142, 137, 128,
19 155, 182, 193, 88, 232, 35, 101, 175, 234, 37, 111, 177, 200, 67, 197, 84,
20 252, 31, 33, 99, 165, 244, 7, 9, 27, 45, 119, 153, 176, 203, 70, 202,
21 69, 207, 74, 222, 121, 139, 134, 145, 168, 227, 62, 66, 198, 81, 243, 14,
22 18, 54, 90, 238, 41, 123, 141, 140, 143, 138, 133, 148, 167, 242, 13, 23,
23 57, 75, 221, 124, 132, 151, 162, 253, 28, 36, 108, 180, 199, 82, 246, 1,
24 };
25
26 word8 S[256] = {
27 99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43, 254, 215, 171, 118,
28 202, 130, 201, 125, 250, 89, 71, 240, 173, 212, 162, 175, 156, 164, 114, 192,
29 183, 253, 147, 38, 54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21,
30 4, 199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39, 178, 117,
31 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214, 179, 41, 227, 47, 132,
32 83, 209, 0, 237, 32, 252, 177, 91, 106, 203, 190, 57, 74, 76, 88, 207,
33 208, 239, 170, 251, 67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168,
34 81, 163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16, 255, 243, 210,
35 205, 12, 19, 236, 95, 151, 68, 23, 196, 167, 126, 61, 100, 93, 25, 115,
36 96, 129, 79, 220, 34, 42, 144, 136, 70, 238, 184, 20, 222, 94, 11, 219,
37 224, 50, 58, 10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121,
38 231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234, 101, 122, 174, 8,
39 186, 120, 37, 46, 28, 166, 180, 198, 232, 221, 116, 31, 75, 189, 139, 138,
40 112, 62, 181, 102, 72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158,
41 225, 248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206, 85, 40, 223,
42 140, 161, 137, 13, 191, 230, 66, 104, 65, 153, 45, 15, 176, 84, 187, 22,
43 };
44
45 word8 Si[256] = {
46 82, 9, 106, 213, 48, 54, 165, 56, 191, 64, 163, 158, 129, 243, 215, 251,
47 124, 227, 57, 130, 155, 47, 255, 135, 52, 142, 67, 68, 196, 222, 233, 203,
48 84, 123, 148, 50, 166, 194, 35, 61, 238, 76, 149, 11, 66, 250, 195, 78,
49 8, 46, 161, 102, 40, 217, 36, 178, 118, 91, 162, 73, 109, 139, 209, 37,
50 114, 248, 246, 100, 134, 104, 152, 22, 212, 164, 92, 204, 93, 101, 182, 146,
51 108, 112, 72, 80, 253, 237, 185, 218, 94, 21, 70, 87, 167, 141, 157, 132,
52 144, 216, 171, 0, 140, 188, 211, 10, 247, 228, 88, 5, 184, 179, 69, 6,
53 208, 44, 30, 143, 202, 63, 15, 2, 193, 175, 189, 3, 1, 19, 138, 107,
54 58, 145, 17, 65, 79, 103, 220, 234, 151, 242, 207, 206, 240, 180, 230, 115,
55 150, 172, 116, 34, 231, 173, 53, 133, 226, 249, 55, 232, 28, 117, 223, 110,
56 71, 241, 26, 113, 29, 41, 197, 137, 111, 183, 98, 14, 170, 24, 190, 27,
57 252, 86, 62, 75, 198, 210, 121, 32, 154, 219, 192, 254, 120, 205, 90, 244,
58 31, 221, 168, 51, 136, 7, 199, 49, 177, 18, 16, 89, 39, 128, 236, 95,
59 96, 81, 127, 169, 25, 181, 74, 13, 45, 229, 122, 159, 147, 201, 156, 239,
60 160, 224, 59, 77, 174, 42, 245, 176, 200, 235, 187, 60, 131, 83, 153, 97,
61 23, 43, 4, 126, 186, 119, 214, 38, 225, 105, 20, 99, 85, 33, 12, 125,
62 };
63
64 word8 iG[4][4] = {
65 0x0e, 0x09, 0x0d, 0x0b,
66 0x0b, 0x0e, 0x09, 0x0d,
67 0x0d, 0x0b, 0x0e, 0x09,

```

```

1  0x09, 0x0d, 0x0b, 0x0e,
2  };
3
4  word32 rcon[30] = {
5      0x01,0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
6      0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
7      0xfa, 0xef, 0xc5, 0x91, };
8

```

Exhibit 3-3 Rijndael Algorithm

```

10 /* rijndael-alg-ref.c  v2.0  August '99
11  * Reference ANSI C code
12  * authors: Paulo Barreto
13  *          Vincent Rijmen
14  *
15  * This code is placed in the public domain.
16  */
17
18 #include <stdio.h>
19 #include <stdlib.h>
20
21 #include "rijndael-alg-ref.h"
22
23 #define SC  ((BC - 4) >> 1)
24
25 #include "boxes-ref.dat"
26
27 static word8 shifts[3][4][2] = {
28     0, 0,
29     1, 3,
30     2, 2,
31     3, 1,
32
33     0, 0,
34     1, 5,
35     2, 4,
36     3, 3,
37
38     0, 0,
39     1, 7,
40     3, 5,
41     4, 4
42 };
43
44
45 word8 mul(word8 a, word8 b) {
46     /* multiply two elements of GF(2^m)
47     * needed for MixColumn and InvMixColumn
48     */
49     if (a && b) return Alogtable[(Logtable[a] + Logtable[b])%255];
50     else return 0;
51 }
52
53 void KeyAddition(word8 a[4][MAXBC], word8 rk[4][MAXBC], word8 BC) {
54     /* Exor corresponding text input and round key input bytes
55     */
56     int i, j;
57
58     for(i = 0; i < 4; i++)
59         for(j = 0; j < BC; j++) a[i][j] ^= rk[i][j];
60 }
61
62 void ShiftRow(word8 a[4][MAXBC], word8 d, word8 BC) {
63     /* Row 0 remains unchanged
64     * The other three rows are shifted a variable amount
65     */

```

```

1   word8 tmp[MAXBC];
2   int i, j;
3
4   for(i = 1; i < 4; i++) {
5       for(j = 0; j < BC; j++) tmp[j] = a[i][(j + shifts[SC][i][d]) % BC];
6       for(j = 0; j < BC; j++) a[i][j] = tmp[j];
7   }
8
9
10  void Substitution(word8 a[4][MAXBC], word8 box[256], word8 BC) {
11      /* Replace every byte of the input by the byte at that place
12       * in the nonlinear S-box
13       */
14      int i, j;
15
16      for(i = 0; i < 4; i++)
17          for(j = 0; j < BC; j++) a[i][j] = box[a[i][j]] ;
18  }
19
20  void MixColumn(word8 a[4][MAXBC], word8 BC) {
21      /* Mix the four bytes of every column in a linear way
22       */
23      word8 b[4][MAXBC];
24      int i, j;
25
26      for(j = 0; j < BC; j++)
27          for(i = 0; i < 4; i++)
28              b[i][j] = mul(2,a[i][j])
29                  ^ mul(3,a[(i + 1) % 4][j])
30                  ^ a[(i + 2) % 4][j]
31                  ^ a[(i + 3) % 4][j];
32      for(i = 0; i < 4; i++)
33          for(j = 0; j < BC; j++) a[i][j] = b[i][j];
34  }
35
36  void InvMixColumn(word8 a[4][MAXBC], word8 BC) {
37      /* Mix the four bytes of every column in a linear way
38       * This is the opposite operation of Mixcolumn
39       */
40      word8 b[4][MAXBC];
41      int i, j;
42
43      for(j = 0; j < BC; j++)
44          for(i = 0; i < 4; i++)
45              b[i][j] = mul(0xe,a[i][j])
46                  ^ mul(0xb,a[(i + 1) % 4][j])
47                  ^ mul(0xd,a[(i + 2) % 4][j])
48                  ^ mul(0x9,a[(i + 3) % 4][j]);
49      for(i = 0; i < 4; i++)
50          for(j = 0; j < BC; j++) a[i][j] = b[i][j];
51  }
52
53  int rijndaelKeySched (word8 k[4][MAXKC], int keyBits, int blockBits, word8
54  W[MAXROUNDS+1][4][MAXBC]) {
55      /* Calculate the necessary round keys
56       * The number of calculations depends on keyBits and blockBits
57       */
58      int KC, BC, ROUNDS;
59      int i, j, t, rconpointer = 0;
60      word8 tk[4][MAXKC];
61
62      switch (keyBits) {
63      case 128: KC = 4; break;
64      case 192: KC = 6; break;
65      case 256: KC = 8; break;
66      default : return (-1);
67      }

```

```

1
2     switch (blockBits) {
3         case 128: BC = 4; break;
4         case 192: BC = 6; break;
5         case 256: BC = 8; break;
6         default : return (-2);
7     }
8
9     switch (keyBits >= blockBits ? keyBits : blockBits) {
10        case 128: ROUNDS = 10; break;
11        case 192: ROUNDS = 12; break;
12        case 256: ROUNDS = 14; break;
13        default : return (-3); /* this cannot happen */
14    }
15
16
17    for(j = 0; j < KC; j++)
18        for(i = 0; i < 4; i++)
19            tk[i][j] = k[i][j];
20    t = 0;
21    /* copy values into round key array */
22    for(j = 0; (j < KC) && (t < (ROUNDS+1)*BC); j++, t++)
23        for(i = 0; i < 4; i++) W[t / BC][i][t % BC] = tk[i][j];
24
25    while (t < (ROUNDS+1)*BC) { /* while not enough round key material
26    calculated */
27        /* calculate new values */
28        for(i = 0; i < 4; i++)
29            tk[i][0] ^= S[tk[(i+1)%4][KC-1]];
30        tk[0][0] ^= rcon[rconpointer++];
31
32        if (KC != 8)
33            for(j = 1; j < KC; j++)
34                for(i = 0; i < 4; i++) tk[i][j] ^= tk[i][j-1];
35        else {
36            for(j = 1; j < KC/2; j++)
37                for(i = 0; i < 4; i++) tk[i][j] ^= tk[i][j-1];
38            for(i = 0; i < 4; i++) tk[i][KC/2] ^= S[tk[i][KC/2 - 1]];
39            for(j = KC/2 + 1; j < KC; j++)
40                for(i = 0; i < 4; i++) tk[i][j] ^= tk[i][j-1];
41        }
42        /* copy values into round key array */
43        for(j = 0; (j < KC) && (t < (ROUNDS+1)*BC); j++, t++)
44            for(i = 0; i < 4; i++) W[t / BC][i][t % BC] = tk[i][j];
45    }
46
47    return 0;
48 }
49
50 int rijndaelEncrypt (word8 a[4][MAXBC], int keyBits, int blockBits, word8
51 rk[MAXROUNDS+1][4][MAXBC])
52 {
53     /* Encryption of one block.
54     */
55     int r, BC, ROUNDS;
56
57     switch (blockBits) {
58         case 128: BC = 4; break;
59         case 192: BC = 6; break;
60         case 256: BC = 8; break;
61         default : return (-2);
62     }
63
64     switch (keyBits >= blockBits ? keyBits : blockBits) {
65         case 128: ROUNDS = 10; break;
66         case 192: ROUNDS = 12; break;
67         case 256: ROUNDS = 14; break;

```

```

1  default : return (-3); /* this cannot happen */
2  }
3
4  /* begin with a key addition
5  */
6  KeyAddition(a,rk[0],BC);
7
8      /* ROUNDS-1 ordinary rounds
9  */
10 for(r = 1; r < ROUNDS; r++) {
11     Substitution(a,S,BC);
12     ShiftRow(a,0,BC);
13     MixColumn(a,BC);
14     KeyAddition(a,rk[r],BC);
15 }
16
17 /* Last round is special: there is no MixColumn
18 */
19 Substitution(a,S,BC);
20 ShiftRow(a,0,BC);
21 KeyAddition(a,rk[ROUNDS],BC);
22
23 return 0;
24 }
25
26
27
28 int rijndaelEncryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
29     word8 rk[MAXROUNDS+1][4][MAXBC], int rounds)
30 /* Encrypt only a certain number of rounds.
31  * Only used in the Intermediate Value Known Answer Test.
32  */
33 {
34     int r, BC, ROUNDS;
35
36     switch (blockBits) {
37     case 128: BC = 4; break;
38     case 192: BC = 6; break;
39     case 256: BC = 8; break;
40     default : return (-2);
41     }
42
43     switch (keyBits >= blockBits ? keyBits : blockBits) {
44     case 128: ROUNDS = 10; break;
45     case 192: ROUNDS = 12; break;
46     case 256: ROUNDS = 14; break;
47     default : return (-3); /* this cannot happen */
48     }
49
50     /* make number of rounds sane */
51     if (rounds > ROUNDS) rounds = ROUNDS;
52
53     /* begin with a key addition
54     */
55     KeyAddition(a,rk[0],BC);
56
57     /* at most ROUNDS-1 ordinary rounds
58     */
59     for(r = 1; (r <= rounds) && (r < ROUNDS); r++) {
60         Substitution(a,S,BC);
61         ShiftRow(a,0,BC);
62         MixColumn(a,BC);
63         KeyAddition(a,rk[r],BC);
64     }
65
66     /* if necessary, do the last, special, round:
67     */

```

```

1     if (rounds == ROUNDS) {
2         Substitution(a,S,BC);
3         ShiftRow(a,0,BC);
4         KeyAddition(a,rk[ROUNDS],BC);
5     }
6
7     return 0;
8 }
9
10
11 int rijndaelDecrypt (word8 a[4][MAXBC], int keyBits, int blockBits, word8
12 rk[MAXROUNDS+1][4][MAXBC])
13 {
14     int r, BC, ROUNDS;
15
16     switch (blockBits) {
17     case 128: BC = 4; break;
18     case 192: BC = 6; break;
19     case 256: BC = 8; break;
20     default : return (-2);
21     }
22
23     switch (keyBits >= blockBits ? keyBits : blockBits) {
24     case 128: ROUNDS = 10; break;
25     case 192: ROUNDS = 12; break;
26     case 256: ROUNDS = 14; break;
27     default : return (-3); /* this cannot happen */
28     }
29
30     /* To decrypt: apply the inverse operations of the encrypt routine,
31      *           in opposite order
32      *
33      * (KeyAddition is an involution: it 's equal to its inverse)
34      * (the inverse of Substitution with table S is Substitution with the
35 inverse table of S)
36      * (the inverse of Shiftrow is Shiftrow over a suitable distance)
37      */
38
39     /* First the special round:
40      *   without InvMixColumn
41      *   with extra KeyAddition
42      */
43     KeyAddition(a,rk[ROUNDS],BC);
44     Substitution(a,Si,BC);
45     ShiftRow(a,1,BC);
46
47     /* ROUNDS-1 ordinary rounds
48      */
49     for(r = ROUNDS-1; r > 0; r--) {
50         KeyAddition(a,rk[r],BC);
51         InvMixColumn(a,BC);
52         Substitution(a,Si,BC);
53         ShiftRow(a,1,BC);
54     }
55
56     /* End with the extra key addition
57      */
58
59     KeyAddition(a,rk[0],BC);
60
61     return 0;
62 }
63
64
65 int rijndaelDecryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
66 word8 rk[MAXROUNDS+1][4][MAXBC], int rounds)
67 /* Decrypt only a certain number of rounds.

```

```

1  * Only used in the Intermediate Value Known Answer Test.
2  * Operations rearranged such that the intermediate values
3  * of decryption correspond with the intermediate values
4  * of encryption.
5  */
6  {
7      int r, BC, ROUNDS;
8
9      switch (blockBits) {
10     case 128: BC = 4; break;
11     case 192: BC = 6; break;
12     case 256: BC = 8; break;
13     default : return (-2);
14     }
15
16     switch (keyBits >= blockBits ? keyBits : blockBits) {
17     case 128: ROUNDS = 10; break;
18     case 192: ROUNDS = 12; break;
19     case 256: ROUNDS = 14; break;
20     default : return (-3); /* this cannot happen */
21     }
22
23
24     /* make number of rounds sane */
25     if (rounds > ROUNDS) rounds = ROUNDS;
26
27     /* First the special round:
28      * without InvMixColumn
29      * with extra KeyAddition
30      */
31     KeyAddition(a,rk[ROUNDS],BC);
32     Substitution(a,Si,BC);
33     ShiftRow(a,l,BC);
34
35     /* ROUNDS-1 ordinary rounds
36      */
37     for(r = ROUNDS-1; r > rounds; r--) {
38         KeyAddition(a,rk[r],BC);
39         InvMixColumn(a,BC);
40         Substitution(a,Si,BC);
41         ShiftRow(a,l,BC);
42     }
43
44     if (rounds == 0) {
45         /* End with the extra key addition
46          */
47         KeyAddition(a,rk[0],BC);
48     }
49
50     return 0;
51 }
52

```

3.1.2. Privacy Procedures

Exhibit 3-4 Header for ESP

```

55 /* "esp.h" Header for ESP. */
56 #ifndef ESP_HEADER
57 #define ESP_HEADER
58
59 #define KEYLENGTH 16 /* octets */
60
61 /* external interface declarations */
62 void ESP_privacykey(unsigned char key[KEYLENGTH]);
63 void

```

```

1  ESP_maskbits(unsigned char *fresh,
2                int freshsize,
3                unsigned char *buf,
4                unsigned long bit_offset,
5                unsigned long bit_count);
6  void
7  ESP_AES(   unsigned char key[KEYLENGTH],
8            unsigned char *fresh,
9            int freshsize,
10           unsigned char *buf,
11           unsigned long bit_offset,
12           unsigned long bit_count);
13
14 #endif
15
16 Exhibit 3-5 ESP_keysched and ESP_maskbits


---


17 /* "esp.c" */
18
19 #include "esp.h"
20
21 #include "rijndael-alg-ref.h"
22 #define BLOCKLENGTH 16 /* octets */
23
24 /* internal storage */
25
26 /* ESP_privacykeyschedule consists of the array rk[ ][ ]. */
27
28 static
29 unsigned char rk[MAXROUNDS+1][4][MAXBC]; /* rijndael Key Schedule */
30
31 /* Schedule key in internal storage. */
32
33 void
34 ESP_privacykey(unsigned char key[KEYLENGTH])
35 {
36     unsigned char k[4][MAXKC];
37     int i;
38
39     /* reshape key for rijndael */
40     for (i = 0; i < KEYLENGTH; ++i)
41         k[i%4][i/4] = key[i];
42     rijndaelKeySched(k, KEYLENGTH*8, BLOCKLENGTH*8, rk);
43 }
44
45
46 /* encrypt/decrypt a buffer of data or output an encryption mask */
47
48 void
49 ESP_maskbits(unsigned char *fresh,
50              int freshsize,
51              unsigned char *buf,
52              unsigned long bit_offset,
53              unsigned long bit_count)
54 {
55     int i;
56     unsigned long counter;
57     unsigned char *bptr;
58     unsigned char b[4][MAXBC], offset, mask, bit_mask, mask_size, last_mask;
59
60     if (bit_count <= 0)
61         return;
62
63     offset = (unsigned char)(bit_offset % 8);
64
65     /* point to first byte to be changed */

```

```

1  bptr = buf + (bit_offset/8);
2
3  for (counter = 0; bit_count > 0; ++counter)
4  {
5      /* initialise buffer with copies of fresh and counter */
6      for (i = 0; i < freshsize; ++i)
7          b[i%4][i/4] = fresh[i];
8      for (/* leftover i */ ; i < BLOCKLENGTH; ++i)
9          /* counter MSB first */
10         b[i%4][i/4] = (unsigned char)(counter >> ((3 - (i%4)) * 8));
11
12     /* run Rijndael */
13     rijndaelEncrypt(b, KEYLENGTH*8, BLOCKLENGTH*8, rk);
14
15     /* set up to use the first bits of the Rijndael buffer */
16     if (offset != 0)
17     {
18         /* set last mask octet to zero */
19         last_mask = 0;
20
21         /* set first mask and its size */
22         mask_size = 8 - offset;
23         bit_mask = 0xff >> offset;
24     }
25     else
26     {
27         mask_size = 8;
28         bit_mask = 0xff;
29     }
30
31     /* adjust for short remaining bit count */
32     if (bit_count < mask_size)
33     {
34         bit_mask &= 0xff << (mask_size - bit_count);
35         mask_size = (unsigned char)bit_count;
36     }
37
38     /* XOR mask into buffer */
39     for (i = 0; i < BLOCKLENGTH; ++i)
40     {
41         if (offset != 0)
42         {
43             mask = last_mask << (8 - offset);
44             last_mask = b[i%4][i/4];
45             mask |= last_mask >> offset;
46         }
47         else
48             mask = b[i%4][i/4];
49         *bptr++ ^= mask & bit_mask;
50         bit_count -= mask_size;
51         if (bit_count <= 0)
52             return;
53
54         /* set next mask and its size */
55         if (bit_count > 7)
56         {
57             mask_size = 8;
58             bit_mask = 0xff;
59         }
60         else
61         {
62             mask_size = (unsigned char)bit_count;
63             bit_mask = 0xff << (8 - mask_size);
64         }
65     }
66
67     /* use the last bits in the Rijndael buffer, if any */

```

```

1         if (offset != 0)
2         {
3             *bptr ^= (last_mask << (8 - offset)) & bit_mask;
4             if (mask_size > offset)
5                 mask_size = offset;
6             bit_count -= mask_size;
7             if (bit_count <= 0)
8                 return;
9         }
10    }
11 }
12
13 void
14 ESP_AES( unsigned char key[KEYLENGTH],
15          unsigned char *fresh,
16          int freshsize,
17          unsigned char *buf,
18          unsigned long bit_offset,
19          unsigned long bit_count)
20 {
21     ESP_privacykey(key);
22     ESP_maskbits(fresh, freshsize, buf, bit_offset, bit_count);
23 }
24

```

3.1.3. KeyStrengthRedAlg Function

Exhibit 3-6 KeyStrengthRedAlg Function Header

```

27 #ifndef KEYSRA
28 #define KEYSRA
29
30 void KeyStrengthRedAlg(
31     /* input */ int KeyLength, /* in octets */
32     /* input */ unsigned char *OriginalKey,
33     /* input */ int SaltLength, /* in octets */
34     /* input */ unsigned char *Salt,
35     /* input */ int KeyEntropy, /* in octets */
36     /* output */ unsigned char *RedStrengthKey
37     /* KeyLength octets in length */
38 );
39 #endif
40

```

Exhibit 3-7 KeyStrengthRedAlg Function Code

```

42 /* Key Strength Reduction Algorithm: "KeySRA.c"*/
43
44 #include "sha.h"
45
46 void KeyStrengthRedAlg(
47     /* input */ int KeyLength, /* in octets */
48     /* input */ unsigned char *OriginalKey,
49     /* input */ int SaltLength, /* in octets */
50     /* input */ unsigned char *Salt,
51     /* input */ int KeyEntropy, /* in octets */
52     /* output */ unsigned char *RedStrengthKey
53     /* Key_Length octets in length */
54 )
55 {
56     int i;
57     SHA_INFO s;
58     unsigned char intermediateKey[DIGEST_LENGTH];
59
60     if (KeyLength > DIGEST_LENGTH)
61         KeyLength = DIGEST_LENGTH;

```

```

1
2     if (KeyEntropy > KeyLength)
3         KeyEntropy = KeyLength;
4
5     shaInitial(&s);
6     shaUpdate(&s, OriginalKey, 0, KeyLength*8);
7     shaUpdate(&s, Salt, 0, SaltLength*8);
8     shaFinal(&s);
9
10    for (i = 0; i < DIGEST_LENGTH - KeyEntropy; ++i)
11        intermediateKey[i] = 0;
12    for ( ; i < DIGEST_LENGTH; ++i)
13        intermediateKey[i] = s.digest[i];
14
15    shaInitial(&s);
16    shaUpdate(&s, intermediateKey, 0, DIGEST_LENGTH*8);
17    shaUpdate(&s, Salt, 0, SaltLength*8);
18    shaFinal(&s);
19
20    for (i = 0; i < KeyLength; ++i)
21        RedStrengthKey[i] = s.digest[i];
22 }
23

```

3.2. Authentication

3.2.1. SHA-1

Exhibit 3-8 SHA-1 Header

```

27 /* "sha.h" */
28
29 #ifndef SHA_H
30 #define SHA_H
31
32 /* header for SHA and related procedures */
33 #define WORD unsigned long
34 #define DIGEST_LENGTH 20
35 #define DIGEST_LENGTH_256 32
36 typedef struct {
37     unsigned char digest[DIGEST_LENGTH_256]; /* Message digest */
38     WORD          count[2]; /* count of bits */
39     WORD          data[16]; /* data buffer */
40 }
41     SHA_INFO;
42
43 void shaInitial(SHA_INFO *sha_info);
44 void shaUpdate(SHA_INFO *sha_info,
45               unsigned char *buffer,
46               unsigned long offset,
47               unsigned long count);
48 void shaFinal(SHA_INFO *sha_info);
49
50 #endif

```

Exhibit 3-9 SHA-1 Code

```

53 /* "sha.c" */
54
55 #include "sha.h"
56
57 static unsigned long A, B, C, D, E;
58

```

```

1  #define K1    0x5a827999
2  #define K2    0x6ed9eba1
3  #define K3    0x8f1bbcdc
4  #define K4    0xca62c1d6
5
6  #define S(a,n) ((a << n) | (a >> (32-n)))
7
8  static
9  unsigned char SHA_IV[20] = { 0x67, 0x45, 0x23, 0x01, 0xef, 0xcd, 0xab, 0x89,
10                             0x98, 0xba, 0xdc, 0xfe, 0x10, 0x32, 0x54, 0x76,
11                             0xc3, 0xd2, 0xe1, 0xf0 };
12
13 /* SHA ft(B,C,D) + Kt */
14
15 static unsigned long ftk(int t)
16 {
17     if (t < 20)
18         return( ((B & C) | (~B & D)) + K1 );
19     else if (t < 40)
20         return( (B ^ C ^ D) + K2 );
21     else if (t < 60)
22         return( ((B & C) | (B & D) | (C & D)) + K3 );
23     else
24         return( (B ^ C ^ D) + K4 );
25 }
26
27 /* the 80 rounds of SHA */
28
29 static
30 void shaHash(SHA_INFO *sha_info)
31 {
32     unsigned long t, A0, B0, C0, D0, E0, W[16];
33     int i, s;
34
35     /* set the temporary digest values from the current digest,
36        using shifts to ensure machine-independence */
37
38     A = (unsigned long)sha_info->digest[0] << 24;
39     A += (unsigned long)sha_info->digest[1] << 16;
40     A += (unsigned long)sha_info->digest[2] << 8;
41     A += (unsigned long)sha_info->digest[3];
42     B = (unsigned long)sha_info->digest[4] << 24;
43     B += (unsigned long)sha_info->digest[5] << 16;
44     B += (unsigned long)sha_info->digest[6] << 8;
45     B += (unsigned long)sha_info->digest[7];
46     C = (unsigned long)sha_info->digest[8] << 24;
47     C += (unsigned long)sha_info->digest[9] << 16;
48     C += (unsigned long)sha_info->digest[10] << 8;
49     C += (unsigned long)sha_info->digest[11];
50     D = (unsigned long)sha_info->digest[12] << 24;
51     D += (unsigned long)sha_info->digest[13] << 16;
52     D += (unsigned long)sha_info->digest[14] << 8;
53     D += (unsigned long)sha_info->digest[15];
54     E = (unsigned long)sha_info->digest[16] << 24;
55     E += (unsigned long)sha_info->digest[17] << 16;
56     E += (unsigned long)sha_info->digest[18] << 8;
57     E += (unsigned long)sha_info->digest[19];
58
59     /* save A-E */
60
61     A0 = A;
62     B0 = B;
63     C0 = C;
64     D0 = D;
65     E0 = E;
66
67     /* move the data into the first 16 words of W */

```

```

1
2   for (i = 0; i < 16; i++)
3       W[i] = sha_info->data[i];
4
5   /* perform the 80 rounds, using the "alternate method" in which
6      the later values of W are computed in place */
7
8   for (i = 0; i < 80; i++)
9   {
10      s = i & 0x0f;
11
12      if (i >= 16)
13      {
14          t = W[(i-3) & 0x0f] ^ W[(i-8) & 0x0f] ^
15             W[(i-14)&0x0f] ^ W[s];
16          W[s] = S(t,1);
17      }
18
19      t = S(A,5) + ftk(i) + E + W[s];
20      E = D;
21      D = C;
22      C = S(B,30);
23      B = A;
24      A = t;
25  }
26
27  /* add in the original values of A-E */
28
29  A += A0;
30  B += B0;
31  C += C0;
32  D += D0;
33  E += E0;
34
35  /* save resulting digest, again using shifts to ensure
36     machine independence */
37
38  sha_info->digest[0] = (unsigned char)(A >> 24);
39  sha_info->digest[1] = (unsigned char)((A >> 16) & 0xff);
40  sha_info->digest[2] = (unsigned char)((A >> 8) & 0xff);
41  sha_info->digest[3] = (unsigned char)(A & 0xff);
42  sha_info->digest[4] = (unsigned char)(B >> 24);
43  sha_info->digest[5] = (unsigned char)((B >> 16) & 0xff);
44  sha_info->digest[6] = (unsigned char)((B >> 8) & 0xff);
45  sha_info->digest[7] = (unsigned char)(B & 0xff);
46  sha_info->digest[8] = (unsigned char)(C >> 24);
47  sha_info->digest[9] = (unsigned char)((C >> 16) & 0xff);
48  sha_info->digest[10] = (unsigned char)((C >> 8) & 0xff);
49  sha_info->digest[11] = (unsigned char)(C & 0xff);
50  sha_info->digest[12] = (unsigned char)(D >> 24);
51  sha_info->digest[13] = (unsigned char)((D >> 16) & 0xff);
52  sha_info->digest[14] = (unsigned char)((D >> 8) & 0xff);
53  sha_info->digest[15] = (unsigned char)(D & 0xff);
54  sha_info->digest[16] = (unsigned char)(E >> 24);
55  sha_info->digest[17] = (unsigned char)((E >> 16) & 0xff);
56  sha_info->digest[18] = (unsigned char)((E >> 8) & 0xff);
57  sha_info->digest[19] = (unsigned char)(E & 0xff);
58
59  /* clear the data so that further updates can be added in */
60
61  for (i = 0; i < 16; i++)
62      sha_info->data[i] = 0;
63  }
64
65  /* initialize sha_info */
66
67  void shaInitial(SHA_INFO *sha_info)

```

```

1  {
2      int i;
3
4      /* set digest to its initial value. Done one char at a time
5         to ensure machine independence. */
6
7      for (i = 0; i < 20; i++)
8          sha_info->digest[i] = SHA_IV[i];
9
10     /* clear data so that updates can be added in */
11
12     for (i = 0; i < 16; i++)
13         sha_info->data[i] = 0;
14
15     /* set bit count to zero */
16
17     sha_info->count[0] = sha_info->count[1] = 0;
18 }
19
20 /* update the digest using additional message data */
21
22 void shaUpdate(SHA_INFO *sha_info,
23               unsigned char *buffer,
24               unsigned long offset,
25               unsigned long count)
26 {
27     unsigned long data_count, t, mask_size;
28     unsigned char *bptr, c, last, mask;
29
30     /* enter the message data into the data buffer. When the buffer
31        is full (512 bits entered, update the digest and clear the
32        data buffer for the next update. */
33
34     data_count = sha_info->count[1]%512;
35     bptr = buffer + (offset/8);
36
37     /* first fill the current octet of the buffer, so that
38        the bit offset into the buffer is a multiple of 8 */
39     last = *bptr++;
40     if (data_count%8)
41     {
42         /* get a full byte from the buffer */
43         c = last;
44         last = *bptr++;
45         if (offset%8)
46         {
47             c <<= (offset%8);
48             c += last >> (8 - (offset%8));
49         }
50
51         /* set mask to fill the remaining bits of the octet */
52         mask_size = 8 - (data_count%8);
53         mask = 0xff << (data_count%8);
54
55         /* adjust for short count */
56         if (count < mask_size)
57         {
58             mask <<= (mask_size-count);
59             mask_size = count;
60         }
61
62         /* store the bits */
63         c = (c & mask) >> (data_count%8);
64         sha_info->data[data_count/32] += (unsigned long)c << 8*(3 -
65 ((data_count%32)/8));
66
67         /* update count */

```

```

1      t = sha_info->count[1];
2      sha_info->count[1] += mask_size;
3      if (sha_info->count[1] < t)
4          sha_info->count[0]++;
5
6      /* if the data buffer is full, update the digest */
7      data_count += mask_size;
8      if (data_count == 512)
9      {
10         shaHash(sha_info);
11         data_count = 0;
12     }
13
14     /* start over with updated offset and count */
15     offset += mask_size;
16     count -= mask_size;
17     bptr = buffer + (offset/8);
18     last = *bptr++;
19 }
20 while (count != 0)
21 {
22     /* get the next full octet from the buffer */
23     c = last;
24     last = *bptr++;
25     if (offset%8)
26     {
27         c <<= (offset%8);
28         c += last >> (8 - (offset%8));
29     }
30
31     /* set mask to a full octet */
32     mask_size = 8;
33     mask = 0xff;
34
35     /* adjust for short count */
36     if (count < mask_size)
37     {
38         mask <<= (mask_size-count);
39         mask_size = count;
40     }
41
42     /* store the bits */
43     c &= mask;
44     sha_info->data[data_count/32] += (unsigned long)c << 8*(3 -
45 ((data_count%32)/8));
46
47     /* update count */
48     t = sha_info->count[1];
49     sha_info->count[1] += mask_size;
50     if (sha_info->count[1] < t)
51         sha_info->count[0]++;
52
53     /* if the data buffer is full, update the digest */
54     data_count += mask_size;
55     if (data_count == 512)
56     {
57         shaHash(sha_info);
58         data_count = 0;
59     }
60
61     count -= mask_size;
62 }
63 }
64
65 /* add pad bit of '1', zero fill and bit length, then update the digest */
66
67 void shaFinal(SHA_INFO *sha_info)

```

```

1  {
2      /* add the pad bit of '1' */
3
4      sha_info->data[(sha_info->count[1]%512)/32] +=
5          1L << (31 - (sha_info->count[1]%32));
6
7      /* if the data buffer is full, update the digest */
8
9      if ((sha_info->count[1]%512) == 511)
10         shaHash(sha_info);
11
12     /* if there isn't room for 64 bits of bit length, leave the
13     buffer zero filled to the end, update the digest and clear
14     the buffer. */
15
16     if ((sha_info->count[1]%512) > (512-65))
17         shaHash(sha_info);
18
19     /* put in the bit length */
20
21     sha_info->data[14] = sha_info->count[0];
22     sha_info->data[15] = sha_info->count[1];
23
24     /* update the digest */
25
26     shaHash(sha_info);
27 }
28

```

3.2.2. SHA-256

Exhibit 3-10 SHA-256 Code

```

31 /* "sha256.c" */
32
33 #include "sha.h"
34
35 static unsigned long A, B, C, D, E, F, G, H;
36
37 static
38 unsigned char SHA_IV[32] = {
39     0x6a, 0x09, 0xe6, 0x67,
40     0xbb, 0x67, 0xae, 0x85,
41     0x3c, 0x6e, 0xf3, 0x72,
42     0xa5, 0x4f, 0xf5, 0x3a,
43     0x51, 0x0e, 0x52, 0x7f,
44     0x9b, 0x05, 0x68, 0x8c,
45     0x1f, 0x83, 0xd9, 0xab,
46     0x5b, 0xe0, 0xcd, 0x19
47 };
48
49 static
50 WORD K[64] = {
51     0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1,
52     0x923f82a4, 0xab1c5ed5,
53     0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe,
54     0x9bdc06a7, 0xc19bf174,
55     0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa,
56     0x5cb0a9dc, 0x76f988da,
57     0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,
58     0x06ca6351, 0x14292967,
59     0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb,
60     0x81c2c92e, 0x92722c85,
61     0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624,
62     0xf40e3585, 0x106aa070,

```

```

1      0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a,
2      0x5b9cca4f, 0x682e6ff3,
3      0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb,
4      0xbef9a3f7, 0xc67178f2
5  };
6
7  WORD ROTR( WORD x, WORD n)
8  { return ( x >> n ) | ( x << (32-n) ) };
9
10 WORD SHR( WORD x, WORD n)
11 { return ( x >> n ); }
12
13 WORD sigma0( WORD x)
14 { return ( ROTR(x,7) ^ ROTR(x,18) ^ SHR(x,3) ); }
15
16 WORD sigma1( WORD x)
17 { return ( ROTR(x,17) ^ ROTR(x,19) ^ SHR(x,10) ); }
18
19 WORD SIGMA0( WORD x)
20 { return ( ROTR(x,2) ^ ROTR(x,13) ^ ROTR(x,22) ); }
21
22 WORD SIGMA1( WORD x)
23 { return ( ROTR(x,6) ^ ROTR(x,11) ^ ROTR(x,25) ); }
24
25 WORD Ch( WORD x, WORD y, WORD z)
26 { return ( (x & y) ^ ( (~x) & z ) ); }
27
28 WORD Maj( WORD x, WORD y, WORD z)
29 { return ( (x & y) ^ ( x & z ) ^ (y & z) ); }
30
31 static
32 void sha256Hash(SHA_INFO *sha_info)
33 {
34     WORD T1, T2, A0, B0, C0, D0, E0, F0, G0, H0, W[64];
35     int i, s;
36
37     /* set the temporary digest values from the current digest,
38        using shifts to ensure machine-independence */
39
40     A = (unsigned long)sha_info->digest[0] << 24;
41     A += (unsigned long)sha_info->digest[1] << 16;
42     A += (unsigned long)sha_info->digest[2] << 8;
43     A += (unsigned long)sha_info->digest[3];
44     B = (unsigned long)sha_info->digest[4] << 24;
45     B += (unsigned long)sha_info->digest[5] << 16;
46     B += (unsigned long)sha_info->digest[6] << 8;
47     B += (unsigned long)sha_info->digest[7];
48     C = (unsigned long)sha_info->digest[8] << 24;
49     C += (unsigned long)sha_info->digest[9] << 16;
50     C += (unsigned long)sha_info->digest[10] << 8;
51     C += (unsigned long)sha_info->digest[11];
52     D = (unsigned long)sha_info->digest[12] << 24;
53     D += (unsigned long)sha_info->digest[13] << 16;
54     D += (unsigned long)sha_info->digest[14] << 8;
55     D += (unsigned long)sha_info->digest[15];
56     E = (unsigned long)sha_info->digest[16] << 24;
57     E += (unsigned long)sha_info->digest[17] << 16;
58     E += (unsigned long)sha_info->digest[18] << 8;
59     E += (unsigned long)sha_info->digest[19];
60     F = (unsigned long)sha_info->digest[20] << 24;
61     F += (unsigned long)sha_info->digest[21] << 16;
62     F += (unsigned long)sha_info->digest[22] << 8;
63     F += (unsigned long)sha_info->digest[23];
64     G = (unsigned long)sha_info->digest[24] << 24;
65     G += (unsigned long)sha_info->digest[25] << 16;
66     G += (unsigned long)sha_info->digest[26] << 8;
67     G += (unsigned long)sha_info->digest[27];

```

```

1      H = (unsigned long)sha_info->digest[28] << 24;
2      H += (unsigned long)sha_info->digest[29] << 16;
3      H += (unsigned long)sha_info->digest[30] << 8;
4      H += (unsigned long)sha_info->digest[31];
5
6      /* save A-H */
7
8      A0 = A;
9      B0 = B;
10     C0 = C;
11     D0 = D;
12     E0 = E;
13     F0 = F;
14     G0 = G;
15     H0 = H;
16
17     /* move the data into the first 16 words of W */
18
19     for (i = 0; i < 16; i++)
20         W[i] = sha_info->data[i];
21
22     for (i = 16; i<64; i++)
23         W[i] = sigma1(W[i-2]) + W[i-7] + sigma0(W[i-15]) + W[i-16];
24
25     /* perform the 64 rounds, using the "alternate method" in which
26        the later values of W are computed in place */
27
28     for (i = 0; i < 64; i++)
29     {
30         T1 = H + SIGMA1(E) + Ch(E,F,G) + K[i] + W[i];
31         T2 = SIGMA0(A) + Maj(A,B,C);
32         H=G;
33         G=F;
34         F=E;
35         E=D + T1;
36         D=C;
37         C=B;
38         B=A;
39         A=T1 + T2;
40     }
41
42     /* add in the original values of A-H */
43
44     A += A0;
45     B += B0;
46     C += C0;
47     D += D0;
48     E += E0;
49     F += F0;
50     G += G0;
51     H += H0;
52
53     /* save resulting digest, again using shifts to ensure
54        machine independence */
55
56     sha_info->digest[0] = (unsigned char)(A >> 24);
57     sha_info->digest[1] = (unsigned char)((A >> 16) & 0xff);
58     sha_info->digest[2] = (unsigned char)((A >> 8) & 0xff);
59     sha_info->digest[3] = (unsigned char)(A & 0xff);
60     sha_info->digest[4] = (unsigned char)(B >> 24);
61     sha_info->digest[5] = (unsigned char)((B >> 16) & 0xff);
62     sha_info->digest[6] = (unsigned char)((B >> 8) & 0xff);
63     sha_info->digest[7] = (unsigned char)(B & 0xff);
64     sha_info->digest[8] = (unsigned char)(C >> 24);
65     sha_info->digest[9] = (unsigned char)((C >> 16) & 0xff);
66     sha_info->digest[10] = (unsigned char)((C >> 8) & 0xff);
67     sha_info->digest[11] = (unsigned char)(C & 0xff);

```

```

1  sha_info->digest[12] = (unsigned char)(D >> 24);
2  sha_info->digest[13] = (unsigned char)((D >> 16) & 0xff);
3  sha_info->digest[14] = (unsigned char)((D >> 8) & 0xff);
4  sha_info->digest[15] = (unsigned char)(D & 0xff);
5  sha_info->digest[16] = (unsigned char)(E >> 24);
6  sha_info->digest[17] = (unsigned char)((E >> 16) & 0xff);
7  sha_info->digest[18] = (unsigned char)((E >> 8) & 0xff);
8  sha_info->digest[19] = (unsigned char)(E & 0xff);
9  sha_info->digest[20] = (unsigned char)(F >> 24);
10 sha_info->digest[21] = (unsigned char)((F >> 16) & 0xff);
11 sha_info->digest[22] = (unsigned char)((F >> 8) & 0xff);
12 sha_info->digest[23] = (unsigned char)(F & 0xff);
13 sha_info->digest[24] = (unsigned char)(G >> 24);
14 sha_info->digest[25] = (unsigned char)((G >> 16) & 0xff);
15 sha_info->digest[26] = (unsigned char)((G >> 8) & 0xff);
16 sha_info->digest[27] = (unsigned char)(G & 0xff);
17 sha_info->digest[28] = (unsigned char)(H >> 24);
18 sha_info->digest[29] = (unsigned char)((H >> 16) & 0xff);
19 sha_info->digest[30] = (unsigned char)((H >> 8) & 0xff);
20 sha_info->digest[31] = (unsigned char)(H & 0xff);
21
22 /* clear the data so that further updates can be added in */
23
24 for (i = 0; i < 16; i++)
25     sha_info->data[i] = 0;
26 }
27
28 /* initialize sha_info */
29
30 void sha256Initial(SHA_INFO *sha_info)
31 {
32     int i;
33
34     /* set digest to its initial value. Done one char at a time
35        to ensure machine independence. */
36
37     for (i = 0; i < DIGEST_LENGTH_256; i++)
38         sha_info->digest[i] = SHA_IV[i];
39
40     /* clear data so that updates can be added in */
41
42     for (i = 0; i < 16; i++)
43         sha_info->data[i] = 0;
44
45     /* set bit count to zero */
46
47     sha_info->count[0] = sha_info->count[1] = 0;
48 }
49
50 /* update the digest using additional message data */
51
52 void sha256Update(SHA_INFO *sha_info,
53                 unsigned char *buffer,
54                 unsigned long offset,
55                 unsigned long count)
56 {
57     unsigned long data_count, t, mask_size;
58     unsigned char *bptr, c, last, mask;
59
60     /* enter the message data into the data buffer. When the buffer
61        is full (512 bits entered, update the digest and clear the
62        data buffer for the next update. */
63
64     data_count = sha_info->count[1]%512;
65     bptr = buffer + (offset/8);
66
67     /* first fill the current octet of the buffer, so that

```

```

1      the bit offset into the buffer is a multiple of 8 */
2      last = *bptr++;
3      if (data_count%8)
4      {
5          /* get a full byte from the buffer */
6          c = last;
7          last = *bptr++;
8          if (offset%8)
9          {
10             c <<= (offset%8);
11             c += last >> (8 - (offset%8));
12         }
13
14         /* set mask to fill the remaining bits of the octet */
15         mask_size = 8 - (data_count%8);
16         mask = 0xff << (data_count%8);
17
18         /* adjust for short count */
19         if (count < mask_size)
20         {
21             mask <<= (mask_size-count);
22             mask_size = count;
23         }
24
25         /* store the bits */
26         c = (c & mask) >> (data_count%8);
27         sha_info->data[data_count/32] += (unsigned long)c << 8*(3 -
28 ((data_count%32)/8));
29
30         /* update count */
31         t = sha_info->count[1];
32         sha_info->count[1] += mask_size;
33         if (sha_info->count[1] < t)
34             sha_info->count[0]++;
35
36         /* if the data buffer is full, update the digest */
37         data_count += mask_size;
38         if (data_count == 512)
39         {
40             sha256Hash(sha_info);
41             data_count = 0;
42         }
43
44         /* start over with updated offset and count */
45         offset += mask_size;
46         count -= mask_size;
47         bptr = buffer + (offset/8);
48         last = *bptr++;
49     }
50     while (count != 0)
51     {
52         /* get the next full octet from the buffer */
53         c = last;
54         last = *bptr++;
55         if (offset%8)
56         {
57             c <<= (offset%8);
58             c += last >> (8 - (offset%8));
59         }
60
61         /* set mask to a full octet */
62         mask_size = 8;
63         mask = 0xff;
64
65         /* adjust for short count */
66         if (count < mask_size)
67         {

```

```

1         mask <<= (mask_size-count);
2         mask_size = count;
3     }
4
5     /* store the bits */
6     c &= mask;
7     sha_info->data[data_count/32] += (unsigned long)c << 8*(3 -
8 ((data_count%32)/8));
9
10    /* update count */
11    t = sha_info->count[1];
12    sha_info->count[1] += mask_size;
13    if (sha_info->count[1] < t)
14        sha_info->count[0]++;
15
16    /* if the data buffer is full, update the digest */
17    data_count += mask_size;
18    if (data_count == 512)
19    {
20        sha256Hash(sha_info);
21        data_count = 0;
22    }
23
24    count -= mask_size;
25 }
26 }
27
28 /* add pad bit of '1', zero fill and bit length, then update the digest */
29
30 void sha256Final(SHA_INFO *sha_info)
31 {
32     /* add the pad bit of '1' */
33
34     sha_info->data[(sha_info->count[1]%512)/32] +=
35         1L << (31 - (sha_info->count[1]%32));
36
37     /* if the data buffer is full, update the digest */
38
39     if ((sha_info->count[1]%512) == 511)
40         sha256Hash(sha_info);
41
42     /* if there isn't room for 64 bits of bit length, leave the
43        buffer zero filled to the end, update the digest and clear
44        the buffer. */
45
46     if ((sha_info->count[1]%512) > (512-65))
47         sha256Hash(sha_info);
48
49     /* put in the bit length */
50
51     sha_info->data[14] = sha_info->count[0];
52     sha_info->data[15] = sha_info->count[1];
53
54     /* update the digest */
55
56     sha256Hash(sha_info);
57 }
58
59

```

60 **3.2.3. Functions f0 and f3**

61 **Exhibit 3-11 f0f3 Function Header**

```

62 /* f0f3.h */
63 /* header for SHA based f0 and f3 functions */

```

```

1
2 #ifndef F0F3_H
3 #define F0F3_H
4
5 #include "sha.h"
6
7 typedef unsigned char    uchar;
8 typedef unsigned short  word16;
9 typedef unsigned long int word32;
10
11 #define L_KEY    16    /*128    bits */
12 #define L_RAND  16    /*128    bits */
13 #define L_FMK   4     /*32    bits */
14 #define L_AMF   2     /*16    bits */
15 #define L_f3K   16    /*128    bits */
16
17 /* function definitions */
18
19 void f0(uchar seed[],uchar fi,uchar Fmk[],uchar buff[]);
20 void f3(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar *f3K);
21
22 #endif
23
24

```

Exhibit 3-12 F0F3 Function Code

```

25 /* f0f3.c: sha based function for f0 and f3 */
26
27 #include <string.h>
28 #include "f0f3.h"
29
30 static uchar counter[8]={0};
31
32 static uchar G[20] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
33                       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
34                       0x00, 0x00, 0x00, 0x2d};
35 static uchar A[20] = { 0x9d, 0xe9, 0xc9, 0xc8, 0xef, 0xd5, 0x78, 0x11,
36                       0x48, 0x23, 0x14, 0x01, 0x90, 0x1f, 0x2d, 0x49,
37                       0x3f, 0x4c, 0x63, 0x65};
38 static uchar B[20] = { 0x75, 0xef, 0xd1, 0x5c, 0x4b, 0x8f, 0x8f, 0x51,
39                       0x4e, 0xf3, 0xbc, 0xc3, 0x79, 0x4a, 0x76, 0x5e,
40                       0x7e, 0xec, 0x45, 0xe0};
41
42 static
43 void modred(uchar *z,int shift,uchar *base);
44
45 /* This function performs the operation of (A*X+B) mod 2^160+2^5+2^3+2^2+1
46 *
47 */
48
49 void whiten(uchar xx[])
50 {
51     uchar z[40];
52     int i, j;
53
54     /* calculate A * X in polynomial form */
55     for (i=0;i<40;i++)
56         z[i]=0;
57
58     for (i=0;i<20;i++)
59     {
60         for (j=0;j<8;j++)
61         {
62             if ((xx[i]<<j) & 0x80)
63                 modred(z,159-(i*8+j),A); /* z^=A<<(159-(i*8+j)) */
64         }
65     }

```

```

1
2
3  /* AX MOD G done as modular reduction for bit 160 to 319 */
4  for (i=0;i<20;i++)
5  {
6      for (j=0;j<8;j++)
7      {
8          if ((z[i]<<j)&0x80)
9              modred(z,159-(i*8+j),G);
10     }
11 }
12
13 /* add B and copy back result */
14 for (i = 0; i < 20; i++)
15     xx[i] = z[i+20] ^ B[i];
16 }
17
18 /* This function perform the operation of shifting 320 bits and XOR.
19  *
20  *
21  */
22
23 static
24 void modred(uchar *z,int shift,uchar *base)
25 {
26     int byteshift, bitshift,i;
27     uchar q[21],yn,ynl;
28
29     for (i=0;i<20;i++)
30         q[i] = base[i];
31     q[20] = 0;
32
33     /* we divide into byte shifting and bit shifting */
34     byteshift = shift / 8;
35     bitshift = shift % 8;
36
37     /* do bit shifting */
38     if (bitshift != 0)
39     {
40         yn = 0;
41         for (i = 0; i <= 20; i++)
42         {
43             ynl = yn;
44             yn = q[i];
45             q[i] >>= 8-bitshift;
46             q[i] |= ynl << bitshift;
47         }
48         /* shift one more byte, since bits have effectively been
49            shifted into the next byte upward */
50         byteshift++;
51     }
52
53     /* z ^= q and send back result in z */
54     for (i = 0; i < 20; i++)
55         z[i+20-byteshift] ^= q[i];
56     if (bitshift != 0)
57         z[40-byteshift] ^= q[20];
58 }
59
60 /* This function performs generation of 64-bit pseudo random number RAND.
61  *
62  *
63  */
64
65 void
66 f0(uchar seed[],uchar fi,uchar Fmk[],uchar buff[])
67 {

```

```

1     SHA_INFO sha_info;
2     uchar buf[64];
3     uchar t;
4     int i;
5
6     shaInitial(&sha_info);
7     for (i = 0; i < L_KEY; i++)
8         sha_info.digest[i] ^= seed[i];
9
10    for (i = 0; i < 64; i++)
11        buf[i] = 0x5c;
12
13    for (i = 0; i < 8; i++)
14    {
15        buf[i] ^= counter[i];
16        buf[i+16] ^= counter[i];
17        buf[i+32] ^= counter[i];
18        buf[i+48] ^= counter[i];
19    }
20
21    buf[11] ^= fi;
22
23    for (i = 0; i < 4; i++)
24        buf[i+12] ^= Fmk[i];
25
26    shaUpdate(&sha_info,buf,0,512);
27
28    /* perform (AX+B)mod G */
29    whiten(sha_info.digest);
30
31    /* get 8 bytes or 64 bits */
32    for (i=0;i<8;i++)
33        buff[i] = sha_info.digest[i];
34
35    /* increment counter */
36    for (i = 7; i >= 0; i--)
37    {
38        t = counter[i];
39        counter[i]++;
40        if (counter[i] > t)
41            break;
42    }
43 }
44
45
46 /* This function performs generation of f3 Output Key f3K.
47  *
48  *
49  */
50
51 void
52 f3(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar *f3K)
53 {
54     SHA_INFO sha_info;
55     uchar j,buf[64];
56     int i;
57
58     for (j = 0; j < 2; j++)
59     {
60         /* NOTE: the following initialization of the sha_info struct can be
61 performed
62         once when K is provisioned, and the results copied into sha_info at
63 the
64         start of this loop. */
65         shaInitial(&sha_info);
66         for (i = 0; i < L_KEY; i++)
67             sha_info.digest[i] ^= K[i];

```

```

1
2     for (i = 0; i < 64; i++)
3         buf[i] = 0x5c;
4
5     for (i = 0; i < 4; i++)
6         buf[i+12] ^= Fmk[i];
7     for (i = 0; i < 16; i++)
8         buf[i+24] ^= RAND[i];
9
10    buf[3] ^= j;
11    buf[11] ^= fi;
12    buf[19] ^= j;
13    buf[35] ^= j;
14    buf[51] ^= j;
15
16    shaUpdate(&sha_info,buf,0,512);
17
18    whiten(sha_info.digest);
19    for (i=0;i<8;i++)
20        f3K[8*j+i] = sha_info.digest[i];
21    }
22 }
23
24

```

3.2.4. GSM Triplet Generation Function fh

Exhibit 3-13 Function fh Header

```

27 /* gsmlway.h */
28
29 #ifndef GSM1WAY
30 #define GSM1WAY
31
32 #include "sha.h"
33
34 typedef unsigned char    uchar;
35 typedef unsigned short   word16;
36 typedef unsigned long int word32;
37
38 #define L_KEY    16    /*128    bits 3G TS 33.105.v.3.0(2000-03)*/
39 #define L_RAND   16    /*128    bits 3G TS 33.105.v.3.0(2000-03)*/
40 #define L_SQN    6     /*48    bits 3G TS 33.105.v.3.0(2000-03)*/
41 #define L_FMK    4     /*32    bits */
42 #define L_CK     16    /*128    bits 3G TS 33.105.v.3.0(2000-03)*/
43
44 typedef struct {
45     unsigned char RAND[16];
46     unsigned char SRES[4];
47     unsigned char Kc[8];
48 } GSM_triplet_type;
49
50 void
51 fh(uchar SSD_A[],uchar SSD_B[],uchar RAND[],uchar fhi,
52     uchar Fmk[],GSM_triplet_type *t);
53
54 #endif
55

```

Exhibit 3-14 Function fh Code

```

57 /* gsmlway.c */
58
59 #include "sha.h"

```

```

1  #include "gsmlway.h"
2
3  static uchar counter[8]={0};
4
5  static uchar G[20] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
6                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
7                        0x00, 0x00, 0x00, 0x2d};
8  static uchar A[20] = { 0x9d, 0xe9, 0xc9, 0xc8, 0xef, 0xd5, 0x78, 0x11,
9                        0x48, 0x23, 0x14, 0x01, 0x90, 0x1f, 0x2d, 0x49,
10                       0x3f, 0x4c, 0x63, 0x65};
11 static uchar B[20] = { 0x75, 0xef, 0xd1, 0x5c, 0x4b, 0x8f, 0x8f, 0x51,
12                       0x4e, 0xf3, 0xbc, 0xc3, 0x79, 0x4a, 0x76, 0x5e,
13                       0x7e, 0xec, 0x45, 0xe0};
14
15 static
16 void modred(uchar *z,int shift,uchar *base);
17
18 /* This function performs the operation of (A*X+B) mod 2^160+2^5+2^3+2^2+1
19  *
20  *
21  */
22
23 static
24 void whiten(uchar xx[])
25 {
26     uchar z[40];
27     int i, j;
28
29     /* calculate A * X in polynomial form */
30     for (i=0;i<40;i++)
31         z[i]=0;
32
33     for (i=0;i<20;i++)
34     {
35         for (j=0;j<8;j++)
36         {
37             if ((xx[i]<<j) & 0x80)
38                 modred(z,159-(i*8+j),A); /* z^=A<<(159-(i*8+j)) */
39         }
40     }
41
42
43     /* AX MOD G done as modular reduction for bit 160 to 319 */
44     for (i=0;i<20;i++)
45     {
46         for (j=0;j<8;j++)
47         {
48             if ((z[i]<<j)&0x80)
49                 modred(z,159-(i*8+j),G);
50         }
51     }
52
53     /* add B and copy back result */
54     for (i = 0; i < 20; i++)
55         xx[i] = z[i+20] ^ B[i];
56 }
57
58
59 /* This function perform the operation of shifting 320 bits and XOR.
60  *
61  *
62  */
63
64 static
65 void modred(uchar *z,int shift,uchar *base)
66 {
67     int byteshift, bitshift,i;

```

```

1   uchar q[21],yn,yn1;
2
3   for (i=0;i<20;i++)
4       q[i] = base[i];
5   q[20] = 0;
6
7   /* we divide into byte shifting and bit shifting */
8   byteshift = shift / 8;
9   bitshift = shift % 8;
10
11  /* do bit shifting */
12  if (bitshift != 0)
13  {
14      yn = 0;
15      for (i = 0; i <= 20; i++)
16          {
17              yn1 = yn;
18              yn = q[i];
19              q[i] >>= 8-bitshift;
20              q[i] |= yn1 << bitshift;
21          }
22      /* shift one more byte, since bits have effectively been
23         shifted into the next byte upward */
24      byteshift++;
25  }
26
27  /* z ^= q and send back result in z */
28  for (i = 0; i < 20; i++)
29      z[i+20-byteshift] ^= q[i];
30  if (bitshift != 0)
31      z[40-byteshift] ^= q[20];
32  }
33
34  /* This function performs generation of a GSM triplet from
35   * SSD and a RAND.
36   *
37   */
38
39  void
40  fh(uchar SSD_A[],uchar SSD_B[],uchar RAND[],uchar fhi,
41     uchar Fmk[],GSM_triplet_type *t)
42  {
43      SHA_INFO sha_info;
44      uchar buf[16], temp[64];
45
46      int i, j;
47
48
49      for (j = 0; j < 2; j++)
50      {
51
52          /* NOTE: the following initialization of the sha_info struct can be
53             performed once when SSD is updated, and the results copied into
54             sha_info at the start of this loop. */
55          shaInitial(&sha_info);
56          for (i = 0; i < 8; i++)
57              {
58                  sha_info.digest[i] ^= SSD_A[i];
59                  sha_info.digest[i+8] ^= SSD_B[i];
60              }
61
62          for (i = 0; i < 64; i++)
63              temp[i] = 0x5c;
64
65          for (i = 0; i < 4; i++)
66              temp[i+12] ^= Fmk[i];
67          for (i = 0; i < 16; i++)

```

```

1         temp[i+24] ^= RAND[i];
2         temp[3] ^= j;
3         temp[11] ^= fhi;
4         temp[19] ^= j;
5         temp[35] ^= j;
6         temp[51] ^= j;
7
8         shaUpdate(&sha_info,temp,0,512);
9
10        whiten(sha_info.digest);
11
12        for (i=0;i<8;i++)
13            buf[8*j+i] = sha_info.digest[i];
14    }
15    for (i = 0; i < 16; i++)
16        t->RAND[i] = RAND[i];
17
18    for (i=0;i<8;i++)
19        t->Kc[i] = buf[i];
20
21    for (i=0;i<4;i++)
22        t->SRES[i] = buf[i+8];
23 }
24

```

3.2.5. CDMA_3G_2G_Conversion Function

Exhibit 3-15 CDMA_3G_2G_Conversion Function Header

```

27 /* twoglway.h */
28
29 #ifndef TWOG1WAY
30 #define TWOG1WAY
31
32 typedef unsigned char    uchar;
33 typedef unsigned short   word16;
34 typedef unsigned long int word32;
35
36 void
37 CDMA_3G_2G_Conversion(uchar CK[],uchar PLCM[],uchar CMEAKEY[]);
38
39 #endif
40

```

Exhibit 3-16 CDMA_3G_2G_Conversion Function Code

```

42 /* twoglway.c */
43
44 #include <string.h>
45 #include "sha.h"
46 #include "twoglway.h"
47
48 /* This function performs generation of 2G privacy and
49 * encryption keys from CK.
50 *
51 */
52
53 void
54 CDMA_3G_2G_Conversion(uchar CK[],uchar PLCM[],uchar CMEAKEY[])
55 {
56     SHA_INFO sha_info;
57     uchar *sha_data = "3G_2GCDMA_conversion";
58     int i;
59
60     shaInitial(&sha_info);
61     shaUpdate(&sha_info,sha_data,0,8*strlen(sha_data));

```

```

1  shaUpdate(&sha_info,CK,0,64);
2  shaFinal(&sha_info);
3
4  for (i = 0; i < 5; i++)
5      PLCM[i] = sha_info.digest[i];
6  for (i = 0; i < 8; i++)
7      CMEKEY[i] = sha_info.digest[i+5];
8  }
9

```

3.3. EHMACHA-1

Exhibit 3-17 EHMACHA Header

```

12 /* ehmacsha.h */
13
14 #ifndef EHMACHA_H
15 #define EHMACHA_H
16 #include "sha.h"
17
18 #define HMAC_IPAD 0x36
19 #define HMAC_OPAD 0x5c
20
21 void ehmacsha_keygen(SHA_INFO *sha_info,
22                     unsigned char *key,
23                     int l_key,
24                     unsigned char pad);
25 void ehmacsha_f(SHA_INFO *sha_info_i,
26                SHA_INFO *sha_info_o,
27                unsigned char *message,
28                unsigned long message_offset,
29                unsigned long message_length,
30                unsigned char *hmac,
31                int l_hmac);
32 void ehmacsha(unsigned char Key[],
33               int l_key,
34               unsigned char *message,
35               unsigned long message_offset,
36               unsigned long message_length,
37               unsigned char *hmac,
38               int l_hmac);
39
40 void ehmacsha_s(SHA_INFO *sha_info_o,
41                unsigned char *message,
42                unsigned long message_offset,
43                unsigned long message_length,
44                unsigned char *ehmac,
45                int l_ehmac);
46
47 void ehmacsha_m(SHA_INFO *sha_info_i, SHA_INFO *sha_info_o,
48                unsigned char *message,
49                unsigned long message_offset,
50                unsigned long message_length,
51                unsigned char *ehmac,
52                int l_ehmac);
53 #endif
54

```

Exhibit 3-18 EHMACHA Code

```

56 /* ehmacsha.c - reference implementation of Enhanced HMAC-SHA-1 */
57
58 #include "ehmacsha.h"
59
60 unsigned char ehmac_pad[] = { 0x80,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
61                               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,

```

```

1             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
2             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
3 unsigned char multiple_block_indicator = { 0x80 };
4
5 /* compute shaHash(key xor pad), the first step in the computation of
6 the inner and outer parts of HMAC. This function must be called twice
7 before calling hmacsha(). The first step sets pad equal to HMAC_IPAD
8 and the third step sets pad equal to HMAC_OPAD. These steps can be
9 precomputed and the sha_info from each step saved for future HMAC
10 computations using the same key. */
11
12 void ehmacsha_keygen(SHA_INFO *sha_info,
13                     unsigned char *key,
14                     int l_key,
15                     unsigned char pad)
16 {
17     unsigned char buf[64], *kptr;
18     int i, l;
19
20     if (l_key > 64)
21     {
22         shaInitial(sha_info);
23         shaUpdate(sha_info, key, 0, 8*l_key);
24         shaFinal(sha_info);
25         kptr = sha_info->digest;
26         l = 20;
27     }
28     else
29     {
30         kptr = key;
31         l = l_key;
32     }
33
34     for (i = 0; i < 64; i++)
35     {
36         buf[i] = pad;
37         if (i < l)
38             buf[i] ^= kptr[i];
39     }
40
41     shaInitial(sha_info);
42     shaUpdate(sha_info, buf, 0, 512);
43 }
44
45 /* Compute EHMAC (typically using IK) for the single block case.
46 hmacsha_keygen() must be called to initialize sha_info_o before
47 calling this function. */
48
49
50 void ehmacsha_s(SHA_INFO *sha_info_o,
51                unsigned char *message,
52                unsigned long message_offset,
53                unsigned long message_length,
54                unsigned char *ehmac,
55                int l_ehmac)
56 {
57     SHA_INFO sha_info;
58     int i;
59
60     sha_info = *sha_info_o;
61
62     shaUpdate(&sha_info, message, message_offset, message_length);
63
64     /* add mandatory 1 and as many zeroes as necessary to fill 512 bits*/
65     shaUpdate(&sha_info, ehmac_pad, 0, 512 - message_length);
66
67     for (i = 0; i < l_ehmac; i++)

```

```

1     ehmac[i] = sha_info.digest[i];
2 }
3
4 /* Compute EHMAL (typically using IK) for the multiple block case.
5 hmacsha_keygen() must be called to initialize sha_info_i & sha_info_o before
6 calling this function. */
7
8 static
9 void ehmacsha_m(SHA_INFO *sha_info_i, SHA_INFO *sha_info_o,
10                unsigned char *message,
11                unsigned long message_offset,
12                unsigned long message_length,
13                unsigned char *ehmac,
14                int l_ehmac)
15 {
16     SHA_INFO sha_info; unsigned char digest[20];
17     int i;
18
19     /* compute the inner part of multiple block ehmacsha */
20     sha_info = *sha_info_i;
21     shaUpdate(&sha_info, message, message_offset, message_length - 351);
22     shaFinal(&sha_info);
23     for(i=0;i<20;i++) digest[i] = sha_info.digest[i];
24
25     /* compute the outer part of multiple block ehmacsha */
26     sha_info = *sha_info_o;
27
28     shaUpdate(&sha_info,digest,0,160); /* add digest of prefix*/
29     shaUpdate(&sha_info, message, message_offset + message_length - 351, 351);
30     /* add suffix*/
31     /* add multiblock indicator bit*/ shaUpdate (&sha_info,
32     &multiple_block_indicator, 0, 1);
33     for (i = 0; i < l_ehmac; i++)
34         ehmac[i] = sha_info.digest[i];
35 }
36
37 /* fast computation of the HMAC of a message, using the results
38 of prior key generation function calls. */
39
40 void ehmacsha_f(SHA_INFO *sha_info_i,
41                SHA_INFO *sha_info_o,
42                unsigned char *message,
43                unsigned long message_offset,
44                unsigned long message_length,
45                unsigned char *ehmac,
46                int l_ehmac)
47 {
48     if (message_length <= 510) /* single block case */
49         ehmacsha_s(sha_info_o,message,message_offset,message_length,
50                   ehmac,l_ehmac);
51     else
52         ehmacsha_m(sha_info_i,sha_info_o,message,message_offset,
53                   message_length,ehmac,l_ehmac);
54 }
55
56 /* complete (slower) computation of the EHMAL of a message, including
57 the key generation function calls. */
58
59 void ehmacsha(unsigned char Key[],
60              int l_key,
61              unsigned char *message,
62              unsigned long message_offset,
63              unsigned long message_length,
64              unsigned char *ehmac,
65              int l_ehmac)
66 {
67     SHA_INFO sha_info_i,sha_info_o;

```

```

1
2     if (message_length <= 510) { /* single block case */
3         ehmacsha_keygen(&sha_info_o,Key,l_key, HMAC_OPAD);
4         ehmacsha_s(&sha_info_o,message,message_offset,message_length,
5             ehmac,l_ehmac);}
6     else { /*multiple block case */
7         ehmacsha_keygen(&sha_info_i,Key,l_key,HMAC_IPAD);
8         ehmacsha_keygen(&sha_info_o,Key,l_key,HMAC_OPAD);
9         ehmacsha_m(&sha_info_i,&sha_info_o,message,message_offset,
10            message_length,ehmac,l_ehmac);}
11 }
12

```

Exhibit 3-19 UMAC_Generation Code

```

14 /* umac.c */
15
16 #include "ehmacsha.h"
17
18 void UMAC_Generation(unsigned char UAK[],
19     int L_UAK,
20     unsigned char MAC[],
21     int l_mac,
22     unsigned char UMAC[])
23 {
24     SHA_INFO sha_info;
25     int i;
26
27     if (l_mac < 1)
28         return;
29     if (l_mac > 64)
30         l_mac = 64;
31
32     /* load the UAK into sha_info. Note that this can be done once when
33        UAK is generated, and the sha_info copied from storage, to save
34        time. */
35
36     shaInitial(&sha_info);
37     for (i = 0; i < L_UAK; i++)
38         sha_info.digest[i] ^= UAK[i];
39
40     ehmacsha_s(&sha_info,MAC,0,8*l_mac,UMAC,l_mac);
41 }
42
43

```

3.4. EHMACHA-SHA-256

Exhibit 3-20 EHMACHA-SHA-256 Header

```

46 /* ehmacsha256.h */
47
48 #ifndef EHMACHA_H
49 #define EHMACHA_H
50 #include "sha.h"
51
52 #define HMAC_IPAD 0x36
53 #define HMAC_OPAD 0x5c
54
55 void ehmacsha256_keygen(SHA_INFO *sha_info,
56     unsigned char *key,
57     int l_key,
58     unsigned char pad);
59 void ehmacsha256(unsigned char Key[],
60     int l_key,

```

```

1         unsigned char *message,
2         unsigned long message_offset,
3         unsigned long message_length,
4         unsigned char *hmac,
5         int l_hmac);
6
7 void ehmacsha256_s(SHA_INFO *sha_info_o,
8                 unsigned char *message,
9                 unsigned long message_offset,
10                unsigned long message_length,
11                unsigned char *ehmac,
12                int l_ehmac);
13
14 void ehmacsha256_m(SHA_INFO *sha_info_i, SHA_INFO *sha_info_o,
15                 unsigned char *message,
16                 unsigned long message_offset,
17                 unsigned long message_length,
18                 unsigned char *ehmac,
19                 int l_ehmac);
20 #endif
21
22 Exhibit 3-21 EHMACHA-SHA-256 Code

```

```

23
24 /* ehmacsha256.c - reference implementation of Enhanced HMAC-SHA-256 */
25
26 #include "ehmacsha256.h"
27
28 unsigned char ehmac_pad[] = { 0x80,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
29                               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
30                               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
31                               0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
32 unsigned char multiple_block_indicator = { 0x80 };
33
34 /* compute shaHash(key xor pad), the first step in the computation of
35 the inner and outer parts of HMAC. This function must be called twice
36 before calling hmacsha(). The first step sets pad equal to HMAC_IPAD
37 and the third step sets pad equal to HMAC_OPAD. These steps can be
38 precomputed and the sha_info from each step saved for future HMAC
39 computations using the same key. */
40
41 void ehmacsha256_keygen(SHA_INFO *sha_info,
42                       unsigned char *key,
43                       int l_key,
44                       unsigned char pad)
45 {
46     unsigned char buf[64],*kptr;
47     int i,l;
48
49     if (l_key > 64)
50     {
51         sha256Initial(sha_info);
52         sha256Update(sha_info,key,0,8*l_key);
53         sha256Final(sha_info);
54         kptr = sha_info->digest;
55         l = 32;
56     }
57     else
58     {
59         kptr = key;
60         l = l_key;
61     }
62
63     for (i = 0; i < 64; i++)
64     {
65         buf[i] = pad;

```

```

1         if (i < 1)
2             buf[i] ^= kpctr[i];
3     }
4
5     sha256Initial(sha_info);
6     sha256Update(sha_info,buf,0,512);
7     sha_info->count[0] = 0;
8     sha_info->count[1] = 0;
9 }
10
11 /* Compute EHMAL (typically using IK) for the single block case.
12 hmacsha_keygen() must be called to initialize sha_info_o before
13 calling this function. */
14
15
16 void ehmacsha256_s(SHA_INFO *sha_info_o,
17                 unsigned char *message,
18                 unsigned long message_offset,
19                 unsigned long message_length,
20                 unsigned char *ehmac,
21                 int l_ehmac)
22 {
23     SHA_INFO sha_info;
24     int i;
25
26     sha_info = *sha_info_o;
27
28     sha256Update(&sha_info,message,message_offset,message_length);
29
30     /* add mandatory 1 and as many zeroes as necessary to fill 512 bits*/
31     sha256Update(&sha_info, ehmac_pad, 0, 512 - message_length);
32
33     for (i = 0; i < l_ehmac; i++)
34         ehmac[i] = sha_info.digest[i];
35 }
36
37 /* Compute EHMAL (typically using IK) for the multiple block case.
38 hmacsha_keygen() must be called to initialize sha_info_i & sha_info_o before
39 calling this function. */
40
41 void ehmacsha256_m(SHA_INFO *sha_info_i, SHA_INFO *sha_info_o,
42                 unsigned char *message,
43                 unsigned long message_offset,
44                 unsigned long message_length,
45                 unsigned char *ehmac,
46                 int l_ehmac)
47 {
48     SHA_INFO sha_info;unsigned char digest[32];
49     int i;
50
51     /* compute the inner part of multiple block ehmacsha */
52     sha_info = *sha_info_i;
53     sha256Update(&sha_info, message, message_offset, message_length - 255);
54     sha256Final(&sha_info);
55     for(i=0;i<32;i++) digest[i] = sha_info.digest[i];
56
57     /* compute the outer part of multiple block ehmacsha */
58     sha_info = *sha_info_o;
59
60     sha256Update(&sha_info,digest,0,256); /* add digest of prefix*/
61     sha256Update(&sha_info, message, message_offset + message_length - 255,
62 255); /* add suffix*/
63
64     /* add multiblock indicator bit*/
65     sha256Update (&sha_info, &multiple_block_indicator, 0, 1);
66     for (i = 0; i < l_ehmac; i++)
67         ehmac[i] = sha_info.digest[i];

```

```
1  }
2
3
4  /* complete (slower) computation of the EHMACH of a message, including
5  the key generation function calls. */
6
7  void ehmacsha256(unsigned char Key[],
8                  int l_key,
9                  unsigned char *message,
10                 unsigned long message_offset,
11                 unsigned long message_length,
12                 unsigned char *ehmac,
13                 int l_ehmac)
14  {
15     SHA_INFO sha_info_i, sha_info_o;
16
17     if (message_length <= 510) { /* single block case */
18         ehmacsha256_keygen(&sha_info_o, Key, l_key, HMAC_OPAD);
19         ehmacsha256_s(&sha_info_o, message, message_offset, message_length,
20                     ehmac, l_ehmac);
21     }
22     else { /*multiple block case */
23         ehmacsha256_keygen(&sha_info_i, Key, l_key, HMAC_IPAD);
24         ehmacsha256_keygen(&sha_info_o, Key, l_key, HMAC_OPAD);
25         ehmacsha256_m(&sha_info_i, &sha_info_o, message, message_offset,
26                     message_length, ehmac, l_ehmac);
27     }
28 }
29
```

4. Test Vectors

4.1. Privacy

4.1.1. Test Program Output

When initialized with the 16 byte ASCII key “Test key 128bits”, with *fresh* of 0x0000000000000001 (represented Most Significant Byte first), and the buffer initialized to zero, the first 41 octets in the buffer should be (in hexadecimal):

Exhibit 4-1 ESP_maskbits Test Output

```

9 bit_offset = 0; bit_count = 328
10
11 ad 23 08 ad 19 1d 93 71 d9 50 f4 d7 a3 a1 48 0c
12 7b 9c ce 3d 62 9a 33 39 61 67 e6 a2 a0 ec 3c c6
13 7b 3a 2a 73 b5 f8 9b 0a 98
14
15 bit_offset = 9; bit_count = 318
16
17 00 56 91 84 56 8c 8e c9 b8 ec a8 7a 6b d1 d0 a4
18 06 3d ce 67 1e b1 4d 19 9c b0 b3 f3 51 50 76 1e
19 63 3d 9d 15 39 da fc 4d 84
20
21 bit_offset = 5; bit_count = 320
22
23 05 69 18 45 68 c8 ec 9b 8e ca 87 a6 bd 1d 0a 40
24 63 dc e6 71 eb 14 d1 99 cb 0b 3f 35 15 07 61 e6
25 33 d9 d1 53 9d af c4 d8 50
26
27 bit_offset = 3; bit_count = 259
28
29 15 a4 61 15 a3 23 b2 6e 3b 2a 1e 9a f4 74 29 01
30 8f 73 99 c7 ac 53 46 67 2c 2c fc d4 54 1d 87 98
31 cc 00 00 00 00 00 00 00 00
32

```

4.1.2. Test Program

Exhibit 4-2 ESP_maskbits Test Program

```

35 #include <stdio.h>
36 #include "esp.h"
37
38 unsigned char *tkey = "Test key 128bits";
39 unsigned char buf[41];
40 unsigned char fresh[8] = { 0, 0, 0, 0, 0, 0, 0, 1 };
41
42 void pause(void)
43 {
44     printf("Press Enter to continue\n");
45     getchar();
46 }
47
48 void main(void)
49 {
50     int i;
51

```

```

1   for (i = 0; i < 41; i++)
2       buf[i] = 0;
3
4   /* bit_offset = 0; bit_count = 8*41 */
5
6   ESP_privacykey(tkey);
7   ESP_maskbits(fresh,8,buf,0,8*41);
8
9   printf("bit_offset = %d; bit_count = %d\n\n",0,8*41);
10
11  for (i = 0; i < 16; i++)
12      printf("%02x ",buf[i]);
13  printf("\n");
14  for (i = 16; i < 32; i++)
15      printf("%02x ",buf[i]);
16  printf("\n");
17  for (i = 32; i < 41; i++)
18      printf("%02x ",buf[i]);
19  printf("\n");
20
21  pause();
22
23  /* bit_offset = 9, bit_count = 7+8*39 */
24
25  printf("bit_offset = %d; bit_count = %d\n\n",9,8*39+6);
26
27  for (i = 0; i < 41; i++)
28      buf[i] = 0;
29
30  ESP_privacykey(tkey);
31  ESP_maskbits(fresh,8,buf,9,8*39+6);
32
33  for (i = 0; i < 16; i++)
34      printf("%02x ",buf[i]);
35  printf("\n");
36  for (i = 16; i < 32; i++)
37      printf("%02x ",buf[i]);
38  printf("\n");
39  for (i = 32; i < 41; i++)
40      printf("%02x ",buf[i]);
41  printf("\n");
42
43  pause();
44
45  /* bit_offset = 5; bit_count = 8*40 */
46
47  printf("bit_offset = %d; bit_count = %d\n\n",5,8*40);
48
49  for (i = 0; i < 41; i++)
50      buf[i] = 0;
51
52  ESP_privacykey(tkey);
53  ESP_maskbits(fresh,8,buf,5,8*40);
54
55  for (i = 0; i < 16; i++)
56      printf("%02x ",buf[i]);
57  printf("\n");
58  for (i = 16; i < 32; i++)
59      printf("%02x ",buf[i]);
60  printf("\n");
61  for (i = 32; i < 41; i++)
62      printf("%02x ",buf[i]);
63  printf("\n");
64
65  pause();
66
67  /* bit_offset = 3; bit_count = 8*32+3 */

```

```

1
2     printf("bit_offset = %d; bit_count = %d\n\n",3,8*32+3);
3
4     for (i = 0; i < 41; i++)
5         buf[i] = 0;
6
7     ESP_privacykey(tkey);
8     ESP_maskbits(fresh,8,buf,3,8*32+3);
9
10    for (i = 0; i < 16; i++)
11        printf("%02x ",buf[i]);
12    printf("\n");
13    for (i = 16; i < 32; i++)
14        printf("%02x ",buf[i]);
15    printf("\n");
16    for (i = 32; i < 41; i++)
17        printf("%02x ",buf[i]);
18    printf("\n");
19
20    pause();
21 }
22

```

4.2. Test Vectors for EHMACHA-1

4.2.1. Test Program Output

```

25
26 test vector for EHMACHA
27 input section
28 IK[16 bytes]:  c1 43 65 25 fa 60 7f 17 92 fc a8 9f b2 a7 bc 4a
29 UAK[16 bytes]:  55 01 c0 20 86 9b 8f ef 7a 33 bb 12 a0 d0 2e 63
30 msg[67 bytes]:
31 "abcdcbdecdefdefgfgfghghighijhijkijklklmklmnlmnomnopnopqopqrpqrsqrx"
32
33 output section
34 ehmac ( 12-bit msg):  f3 61 35 21 91 51 51 5d 4e 5d 57 11 b4 79 62 dd 79 c0 05
35 2b
36 ehmac (510-bit msg):  a0 ea ed 4b 19 9e a9 8c f4 bc 92 89 89 43 b5 e2 28 aa b7
37 5a
38 ehmac (511-bit msg):  02 81 50 67 ac a0 29 77 ae 2e 73 13 fd d6 f9 d0 55 be 37
39 fa
40 ehmac (520-bit msg):  70 3b de d1 34 3d 73 e9 80 e7 6a 22 9b c3 74 cd 43 bb c2
41 e6
42 umac (520-bit msg):  c1 04 54 af 0b 8f 6b 6b 00 b4 32 54 c2 8a 5a 36 37 90 ee
43 16
44
45 Repeat with 8-bit message offset
46
47 ehmac ( 12-bit msg):  be 36 64 30 73 7d be d1 a8 f0 16 6a 8d 20 22 df 38 11 7f
48 c1
49 ehmac (510-bit msg):  97 e3 89 1e ad cb 83 46 22 97 23 80 ac 92 44 36 94 0f 53
50 4d
51 ehmac (511-bit msg):  05 7e 38 e5 68 9e 6c 7c cc 6e 5b 16 fd 1d e5 0d 86 c2 3e
52 50
53 ehmac (520-bit msg):  92 b1 22 c2 7c 4e 15 bb 33 b8 0d bb 07 34 16 62 00 16 25
54 c3
55 umac (520-bit msg):  25 bc e0 92 cc 4e 54 28 3d 62 ab 18 d3 06 2f 41 59 10 50
56 9c

```

4.2.2. Test Program

```

58 /* ehmacetest.c */
59
60 #include <stdio.h>

```

```

1  #include <string.h>
2  #include "ehmacsha.h"
3
4  #define L_KEY    16
5  #define L_UAK   16
6
7  unsigned char *ehmac_test =
8  "abcdcbdecdefdefgefghfghighijhijkiijkljklmklmnlmnomnopnopqopqrpqrsqrx";
9
10 unsigned char IK[]={ 0xc1, 0x43, 0x65, 0x25, 0xfa, 0x60, 0x7f, 0x17,
11                    0x92, 0xfc, 0xa8, 0x9f, 0xb2, 0xa7, 0xbc, 0x4a };
12
13 unsigned char UAK[]={ 0x55, 0x01, 0xc0, 0x20, 0x86, 0x9b, 0x8f, 0xef,
14                    0x7a, 0x33, 0xbb, 0x12, 0xa0, 0xd0, 0x2e, 0x63 };
15
16 static void pause(void)
17 {
18     printf("press any key to continue\n");
19     getchar();
20 }
21
22 void main(void)
23 {
24     unsigned char umac[20];
25     int i;
26
27     printf("\n");
28     printf("\n");
29     printf("\n");
30     printf("test vector for EHMACHA\n");
31     printf("input section\n");
32     printf("IK[%ld bytes]:  ", L_KEY);
33     for(i=0; i<L_KEY; i++)
34         printf("%02x ",IK[i]);
35     printf("\n");
36     printf("UAK[%ld bytes]:  ", L_UAK);
37     for(i=0; i<L_UAK; i++)
38         printf("%02x ",UAK[i]);
39     printf("\n");
40     printf("msg[%ld bytes]:  \"%s\"\n", strlen(ehmac_test), ehmac_test);
41     printf("\n");
42
43     printf("output section\n");
44
45     /* run EHMACHA-SHA on a short messages (12 and 510 bits long) */
46     ehmacsha(IK, 16, ehmac_test, 0, 12,  umac, 20);
47     printf("ehmac ( 12-bit msg):  ");
48     for (i=0; i<20; i++)
49         printf("%02x ", umac[i]);
50     printf("\n");
51     ehmacsha(IK, 16, ehmac_test, 0, 510, umac, 20);
52     printf("ehmac (510-bit msg):  ");
53     for (i=0; i<20; i++)
54         printf("%02x ", umac[i]);
55     printf("\n");
56
57     /* run EHMACHA-SHA on a long message (511 bits long) */
58     /* and verify that mode transition takes place */
59     ehmacsha(IK, 16, ehmac_test, 0, 511, umac, 20);
60     printf("ehmac (511-bit msg):  ");
61     for (i=0; i<20; i++)
62         printf("%02x ", umac[i]);
63     printf("\n");
64
65     /* run EHMACHA on longer-than-buffer msg */
66     ehmacsha(IK, 16, ehmac_test, 0, 520, umac, 20);
67     printf("ehmac (520-bit msg):  ");

```

```

1     for (i=0; i<20; i++)
2         printf("%02x ", umac[i]);
3     printf("\n");
4
5     /* create the UMAC for this result */
6     UMAC_Generation(UAK,L_UAK,umac,20,umac);
7     printf("umac (520-bit msg):  ");
8     for (i=0; i<20; i++)
9         printf("%02x ", umac[i]);
10    printf("\n");
11    printf("\n");
12
13    /* repeat the test with message offset 8 bits */
14    printf("Repeat with 8-bit message offset\n\n");
15
16    /* run EHMAL-SHA on a short messages (12 and 510 bits long) */
17    ehmacsha(IK, 16, ehmac_test, 8, 12,  umac, 20);
18    printf("ehmac ( 12-bit msg):  ");
19    for (i=0; i<20; i++)
20        printf("%02x ", umac[i]);
21    printf("\n");
22    ehmacsha(IK, 16, ehmac_test, 8, 510, umac, 20);
23    printf("ehmac (510-bit msg):  ");
24    for (i=0; i<20; i++)
25        printf("%02x ", umac[i]);
26    printf("\n");
27
28    /* run EHMAL-SHA on a long message (511 bits long) */
29    /* and verify that mode transition takes place      */
30    ehmacsha(IK, 16, ehmac_test, 8, 511, umac, 20);
31    printf("ehmac (511-bit msg):  ");
32    for (i=0; i<20; i++)
33        printf("%02x ", umac[i]);
34    printf("\n");
35
36    /* run EHMAL on longer-than-buffer msg */
37    ehmacsha(IK, 16, ehmac_test, 8, 520, umac, 20);
38    printf("ehmac (520-bit msg):  ");
39    for (i=0; i<20; i++)
40        printf("%02x ", umac[i]);
41    printf("\n");
42
43    /* create the UMAC for this result */
44    UMAC_Generation(UAK,L_UAK,umac,20,umac);
45    printf("umac (520-bit msg):  ");
46    for (i=0; i<20; i++)
47        printf("%02x ", umac[i]);
48    printf("\n");
49    printf("\n");
50
51    pause();
52
53 }

```

4.3. Test Vectors for EHMAL-SHA-256

4.3.1. Test Program Output

```

56 test vector for EHMAL-SHA256
57 input section
58 IK[16 bytes]:  c1 43 65 25 fa 60 7f 17 92 fc a8 9f b2 a7 bc 4a

```

```

1 msg[67 bytes]:
2 "abcdbcdecdefdefgefghfghighijhijkljklmklmnlmnomnopnopqopqrpqrsqrx"
3
4 output section
5 ehmac ( 12-bit msg):  6b 76 ac c9 15 1d 4d 19 3e f8 cd ea 03 26 80 e8 ca b8 54
6 8d f4 77 3e a9 ca 28 e4 f2 f7 ec 33 0c
7 ehmac (510-bit msg):  46 8c fb eb 91 71 0d 31 89 8b 94 e7 2b 45 76 7a 5f fc 7f
8 10 28 a5 e2 e8 8d c6 93 26 46 fd 54 2f
9 ehmac (511-bit msg):  b8 7b 65 3d e8 28 d8 bb 69 fb a2 56 fd 31 89 22 2c 12 f0
10 0a 40 59 d5 5d d8 db c1 0e 18 5b e3 e6
11 ehmac (520-bit msg):  84 e2 70 1a 1a 9d bb 0d ed b5 91 fc 33 04 1b de c1 d6 80
12 57 e1 83 39 3a 18 16 15 98 26 be bl dl
13 Repeat with 8-bit message offset
14
15 ehmac ( 12-bit msg):  d4 e2 ae 19 64 61 88 94 8e 6f 24 53 89 b3 2f 89 6f 79 69
16 3a 5b 2e dd d6 7a 69 0f 8c 00 48 93 21
17 ehmac (510-bit msg):  4b 67 83 2d 45 f2 f2 91 4a 51 ab 73 2a 7f 90 f9 1d 72 30
18 58 3d 1b 14 e1 5b 09 07 46 48 d8 58 ca
19 ehmac (511-bit msg):  8a cd d7 54 36 19 0a 63 43 22 43 a6 ab 61 53 70 08 78 5d
20 4b 66 49 55 ce 9d 89 ab 90 7f 36 af e8
21 ehmac (520-bit msg):  28 d6 46 9a 3a ff 16 c6 77 f6 c0 99 87 41 b5 19 97 bf 31
22 ed 52 fa 0b e7 89 dd ee 04 06 7a 4b 40
23

```

4.3.2. Test Program

```

24
25
26 /* ehmacsha256test.c */
27
28 #include <stdio.h>
29 #include <string.h>
30 #include "ehmacsha256.h"
31
32 #define L_KEY    16
33
34 unsigned char *ehmac_test =
35 "abcdbcdecdefdefgefghfghighijhijkljklmklmnlmnomnopnopqopqrpqrsqrx";
36
37 unsigned char IK[]={ 0xc1, 0x43, 0x65, 0x25, 0xfa, 0x60, 0x7f, 0x17,
38 0x92, 0xfc, 0xa8, 0x9f, 0xb2, 0xa7, 0xbc, 0x4a };
39
40 static void pause(void)
41 {
42     printf("press any key to continue\n");
43     getchar();
44 }
45
46 void main(void)
47 {
48     unsigned char umac[32];
49     int i;
50
51     printf("\n");
52     printf("\n");
53     printf("\n");
54     printf("test vector for EHMACH-SHA256\n");
55     printf("input section\n");
56     printf("IK[%ld bytes]:  ", L_KEY);
57     for(i=0; i<L_KEY; i++)
58         printf("%02x ",IK[i]);
59     printf("\n");
60     printf("msg[%ld bytes]:  \"%s\"\n", strlen(ehmac_test), ehmac_test);
61     printf("\n");
62
63     printf("output section\n");
64
65     /* run EHMACH-SHA256 on a short messages (12 and 510 bits long) */

```

```

1     ehmacsha256(IK, 16, ehmac_test, 0, 12, umac, 32);
2     printf("ehmac ( 12-bit msg): ");
3     for (i=0; i<32; i++)
4         printf("%02x ", umac[i]);
5     printf("\n");
6     ehmacsha256(IK, 16, ehmac_test, 0, 510, umac, 32);
7     printf("ehmac (510-bit msg): ");
8     for (i=0; i<32; i++)
9         printf("%02x ", umac[i]);
10    printf("\n");
11
12    /* run EHMAL-SHA256 on a long message (511 bits long) */
13    /* and verify that mode transition takes place */
14    ehmacsha256(IK, 16, ehmac_test, 0, 511, umac, 32);
15    printf("ehmac (511-bit msg): ");
16    for (i=0; i<32; i++)
17        printf("%02x ", umac[i]);
18    printf("\n");
19
20    /* run EHMAL on longer-than-buffer msg */
21    ehmacsha256(IK, 16, ehmac_test, 0, 520, umac, 32);
22    printf("ehmac (520-bit msg): ");
23    for (i=0; i<32; i++)
24        printf("%02x ", umac[i]);
25    printf("\n");
26
27    /* repeat the test with message offset 8 bits */
28    printf("Repeat with 8-bit message offset\n\n");
29
30    /* run EHMAL-SHA256 on a short messages (12 and 510 bits long) */
31    ehmacsha256(IK, 16, ehmac_test, 8, 12, umac, 32);
32    printf("ehmac ( 12-bit msg): ");
33    for (i=0; i<32; i++)
34        printf("%02x ", umac[i]);
35    printf("\n");
36    ehmacsha256(IK, 16, ehmac_test, 8, 510, umac, 32);
37    printf("ehmac (510-bit msg): ");
38    for (i=0; i<32; i++)
39        printf("%02x ", umac[i]);
40    printf("\n");
41
42    /* run EHMAL-SHA256 on a long message (511 bits long) */
43    /* and verify that mode transition takes place */
44    ehmacsha256(IK, 16, ehmac_test, 8, 511, umac, 32);
45    printf("ehmac (511-bit msg): ");
46    for (i=0; i<32; i++)
47        printf("%02x ", umac[i]);
48    printf("\n");
49
50    /* run EHMAL on longer-than-buffer msg */
51    ehmacsha256(IK, 16, ehmac_test, 8, 520, umac, 32);
52    printf("ehmac (520-bit msg): ");
53    for (i=0; i<32; i++)
54        printf("%02x ", umac[i]);
55    printf("\n");
56
57
58    pause();
59
60 }
61

```

4.4. Test Vectors for One-Way Roaming to 2G Systems

4.4.1. Test Program Output

```

3 test vector for fh
4 input section
5 SSD_A is: ad 1b 5a 15 9b e8 6b 2c
6 SSD_B is: a6 6c 7a e4 0b ba 9b 9d
7 RAND is: 4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
8 fih is: 60
9 Fmk is: 41 48 41 47
10
11 output section
12 fh Kc: 1b 08 ad 36 44 ba 2a 85
13 fh SRES: 92 06 4a d2
14
15
16 test vector for 3G-2G conversion
17 input section
18 CK is: 6e fd d8 32 f6 ff d4 dc a8 4a 54 96 fa 6e 29 93
19
20 output section
21 2G PLCM: 52 16 ad b2 9e
22 2G CMEAKEY: 9d fd d1 45 a9 fe 45 31
23

```

4.4.2. Test Program

```

25 /* test_fh and 3G to 2G conversion */
26
27 #include <stdio.h>
28 #include "gsmlway.h"
29 #include "twoglway.h"
30
31 void pause(void)
32 {
33     printf("Press any key to continue\n");
34     getchar();
35 }
36
37 void main()
38 {
39     uchar K[]={
40         0xad,0x1b,0x5a,0x15,0x9b,0xe8,0x6b,0x2c,
41         0xa6,0x6c,0x7a,0xe4,0x0b,0xba,0x9b,0x9d };
42
43     uchar seed[]={
44         0xb0,0xab,0xb9,0x9d,0x6a,0xc6,0xa7,0x4e,
45         0xb9,0x8e,0xb6,0xc2,0xda,0xb1,0xa5,0x51 };
46
47     uchar Fmk[L_FMK] = { 'A', 'H', 'A', 'G' };
48     uchar RAND[L_RAND] = {
49         0x4b, 0x05, 0x2b, 0x20, 0xe2, 0xa0, 0x6c, 0x8f,
50         0xf7, 0x00, 0xda, 0x51, 0x2b, 0x4e, 0x11, 0x1e };
51     uchar CK[L_CK] = {
52         0x6e, 0xfd, 0xd8, 0x32, 0xf6, 0xff, 0xd4, 0xdc,
53         0xa8, 0x4a, 0x54, 0x96, 0xfa, 0x6e, 0x29, 0x93 };
54
55     GSM_triplet_type triplet;
56     uchar CMEAKEY[8];
57     uchar PLCM[5];
58
59     uchar SQN[L_SQN]={0x00,0x00,0x00,0x00,0x00,0x01};
60     uchar fih;
61

```

```

1     int i;
2
3     fih = 0x60;
4
5     printf("\n");
6     printf("\n");
7     printf("test vector for fh\n");
8     printf("input section\n");
9     printf("SSD_A is: ");
10    for(i=0;i<L_KEY/2;i++)
11        printf("%02x ",K[i]);
12    printf("\n");
13    printf("SSD_B is: ");
14    for(i=0;i<L_KEY/2;i++)
15        printf("%02x ",K[i+8]);
16    printf("\n");
17    printf("RAND is: ");
18    for(i=0;i<L_RAND;i++)
19        printf("%02x ",RAND[i]);
20    printf("\n");
21    printf("fih is:      %02x\n",fih);
22    printf("Fmk is:      ");
23    for(i=0;i<L_FMK;i++)
24        printf("%02x ",Fmk[i]);
25    printf("\n");
26    printf("\n");
27    fh(&K[0],&K[8],RAND,fih,Fmk,&triplet);
28    printf("output section\n");
29    printf("fh Kc:      ");
30    for (i=0;i<8;i++)
31        printf("%02x ",triplet.Kc[i]);
32    printf("\n");
33    printf("fh SRES:  ");
34    for (i=0;i<4;i++)
35        printf("%02x ",triplet.SRES[i]);
36    printf("\n");
37
38    pause();
39    printf("\n");
40    printf("\n");
41    printf("test vector for 3G-2G conversion\n");
42    printf("input section\n");
43    printf("CK is:  ");
44    for(i=0;i<L_KEY;i++)
45        printf("%02x ",CK[i]);
46    printf("\n");
47    printf("\n");
48    CDMA_3G_2G_Conversion(CK,PLCM,CMEAKEKEY);
49    printf("output section\n");
50    printf("2G PLCM:  ");
51    for (i=0;i<5;i++)
52        printf("%02x ",PLCM[i]);
53    printf("\n");
54    printf("2G CMEAKEKEY:  ");
55    for (i=0;i<8;i++)
56        printf("%02x ",CMEAKEKEY[i]);
57    printf("\n");
58
59    pause();
60 }

```

4.5. Functions f0 and f3

4.5.1. Test Program Output

Exhibit 4-3 Functions f0 and f3 Test Output

```
4 test vector for f0:
5 input section
6 seed is:  b0 ab b9 9d 6a c6 a7 4e b9 8e b6 c2 da b1 a5 51
7 fi0 is:    41
8 Fmk is:    41 48 41 47
9
10 output section
11 f0 RAND:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
12
13
14 test vector for f3:
15 input section
16 K is:     ad 1b 5a 15 9b e8 6b 2c a6 6c 7a e4 0b ba 9b 9d
17 fi3 is:    45
18 RAND is:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
19 Fmk is:    41 48 41 47
20
21 output section
22 f3K:     6e fd d8 32 f6 ff d4 dc a8 4a 54 96 fa 6e 29 93
23
```

4.5.2. Test Program

Exhibit 4-4 Functions f0 and f3 Test Program

```

3  /* test_functions f0 and f3 */
4
5  #include <stdio.h>
6  #include "f0f3.h"
7
8  void pause(void)
9  {
10     printf("Press any key to continue\n");
11     getchar();
12 }
13
14 void main()
15 {
16     uchar K[]={0xad,0x1b,0x5a,0x15,0x9b,0xe8,0x6b,0x2c,
17               0xa6,0x6c,0x7a,0xe4,0x0b,0xba,0x9b,0x9d};
18
19     uchar seed[]={0xb0,0xab,0xb9,0x9d,0x6a,0xc6,0xa7,0x4e,
20                 0xb9,0x8e,0xb6,0xc2,0xda,0xb1,0xa5,0x51};
21
22     uchar Fmk[L_FMK] = { 'A', 'H', 'A', 'G' };
23     uchar RAND[L_RAND];
24
25     uchar f3K[L_f3K];
26
27     uchar fi0,fi3;
28
29     uchar buff1[L_RAND/2],buff2[L_RAND/2];
30
31     uchar AMF[2];
32
33     uchar OriginalKey[16] = { 0x52, 0x65, 0x67, 0x69, 0x73, 0x74, 0x72, 0x61,
34                               0x74, 0x69, 0x6f, 0x6e, 0x4d, 0x61, 0x73, 0x74 };
35     uchar Salt[4] = { 0x4d, 0x6f, 0x62, 0x69 };
36
37     int i;
38
39     fi0=0x41;
40     fi3=0x45;
41
42     printf("test vector for f0:\n");
43     printf("input section\n");
44     printf("seed is: ");
45     for(i=0;i<L_KEY;i++)
46         printf("%02x ",seed[i]);
47     printf("\n");
48     printf("fi0 is:      %02x\n",fi0);
49     printf("Fmk is:          ");
50     for (i=0;i<L_FMK;i++)
51         printf("%02x ",Fmk[i]);
52     printf("\n");
53
54     printf("\n");
55     f0(seed,fi0,Fmk,buff1);
56     f0(seed,fi0,Fmk,buff2);
57     printf("output section\n");
58     printf("f0  RAND: ");
59     for (i=0;i<L_RAND/2;i++)
60         printf("%02x ",buff1[i]);
61     for (i=0;i<L_RAND/2;i++)
62         printf("%02x ",buff2[i]);
63     printf("\n");

```

```

1
2     pause();
3
4     /* reuse RAND generated for the subsequent function calls*/
5     for (i=0;i<L_RAND/2;i++)
6         RAND[i] = buff1[i];
7     for (i=0;i<L_RAND/2;i++)
8         RAND[i+L_RAND/2] = buff2[i];
9
10    AMF[0]=0x00;
11    AMF[1]=0x01;
12
13    printf("\n");
14    printf("\n");
15    printf("test vector for f3:\n");
16    printf("input section\n");
17    printf("K is: ");
18    for(i=0;i<L_KEY;i++)
19        printf("%02x ",K[i]);
20    printf("\n");
21    printf("fi3 is:      %02x\n",fi3);
22    printf("RAND is: ");
23    for(i=0;i<L_RAND;i++)
24        printf("%02x ",RAND[i]);
25    printf("\n");
26    printf("Fmk is:      ");
27    for(i=0;i<L_FMK;i++)
28        printf("%02x ",Fmk[i]);
29    printf("\n");
30
31    printf("\n");
32    f3(K,fi3,RAND,Fmk,f3K);
33    printf("output section\n");
34    printf("f3K: ");
35    for (i=0;i<L_f3K;i++)
36        printf("%02x ",f3K[i]);
37    printf("\n");
38
39    pause();
40 }
41
42

```