

1 3GPP2 S.S0078-0

2 Version 1.0

3 Version Date: 12 December 2002



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

26

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>Approved for publication by TSG-S</i>	<i>12-12-02</i>

9

Table of Contents

1

2	1. INTRODUCTION	1
3	1.1. Notations	1
4	1.2. Definitions	1
5	2. PROCEDURES	2
6	2.1. Hash Algorithm	2
7	2.1.1. SHA-1	2
8	2.1.2. SHA-based MAC	3
9	2.1.2.1. MAC Calculation Procedure	3
10	2.1.2.2. UIM-Present MAC (UMAC) Generation Procedure	5
11	2.2. Authentication	6
12	2.2.1. UIM Authentication	6
13	2.2.2. One-Way Roaming to 2G systems	7
14	2.2.2.1. GSM Triplet Generation from SSD	7
15	2.2.2.2. 2G Key Generation from 3G Keys	9
16	2.3. Voice and Data Privacy	10
17	2.3.1. Encryption Key Generation	10
18	2.3.2. Key Strength Reduction	10
19	2.3.3. Enhanced Privacy Algorithm	11
20	2.3.3.1. Algorithm	11
21	2.3.3.2. ESP_privacykey Procedure	11
22	2.3.3.3. ESP_maskbits Procedure	12
23	2.3.3.4. ESP_AES Procedure	14
24	3. REFERENCE IMPLEMENTATIONS	15
25	3.1. Privacy	15
26	3.1.1. Rijndael	15
27	3.1.2. Privacy Procedures	24
28	3.1.3. KeyStrengthRedAlg Function	27
29	3.2. Authentication	28
30	3.2.1. SHA-1	28
31	3.2.2. GSM Triplet Generation Function fh	34
32	3.2.3. CDMA_3G_2G_Conversion Function	37
33	3.3. EHMAL-SHA-1	38
34	4. TEST VECTORS	43
35	4.1. Privacy	43
36	4.1.1. Test Program Output	43
37	4.1.2. Test Program	43

1	4.2. Test Vectors for EHMACHA-1	45
2	4.2.1. Test Program Output	45
3	4.2.2. Test Program	46
4	4.3. Test Vectors for One-Way Roaming to 2G Systems	47
5	4.3.1. Test Program Output	47
6	4.3.2. Test Program	48
7		

List of Exhibits

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20

EXHIBIT 3-1 HEADER FOR RIJNDAEL.....	15
EXHIBIT 3-2 RIJNDAEL BOX DATA	15
EXHIBIT 3-3 RIJNDAEL ALGORITHM	17
EXHIBIT 3-4 HEADER FOR ESP	24
EXHIBIT 3-5 ESP_KEYSCHED AND ESP_MASKBITS	24
EXHIBIT 3-6 KEYSTRENGTHREDALG FUNCTION HEADER.....	27
EXHIBIT 3-7 KEYSTRENGTHREDALG FUNCTION CODE.....	27
EXHIBIT 3-8 SHA-1 HEADER	28
EXHIBIT 3-9 SHA-1 CODE.....	28
EXHIBIT 3-10 FUNCTION FH HEADER.....	34
EXHIBIT 3-11 FUNCTION FH CODE.....	34
EXHIBIT 3-12 CDMA_3G_2G_CONVERSION FUNCTION HEADER.....	37
EXHIBIT 3-13 CDMA_3G_2G_CONVERSION FUNCTION CODE.....	38
EXHIBIT 3-14 EHMAC HEADER.....	38
EXHIBIT 3-15 EHMAC CODE.....	39
EXHIBIT 3-16 UMAC_GENERATION CODE.....	42
EXHIBIT 4-1 RIJNDAEL TEST OUTPUT.....	43
EXHIBIT 4-2 RIJNDAEL TEST PROGRAM.....	43

1. Introduction

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

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.

1.1. Notations

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 ANSI/ISO 9899-1990, "Programming Languages - C"). 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

AND	Bitwise logical AND function.
Internal Stored Data	Stored data that is defined locally within the cryptographic procedures and is not accessible for examination or use outside those procedures.
LSB	Least Significant Bit.
MSB	Most Significant Bit.
OR	Bitwise logical inclusive OR function.
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.

2. Procedures

2.1. Hash Algorithm

2.1.1. SHA-1

The hash function used in this document is SHA-1, defined in FIPS publication FIPS 180-1, "Secure Hash Standard," April 17, 1995. 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 algorithm.

Test vectors for SHA-1 are given in FIPS 180-1.

SHA-1 uses 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 hash function defined in FIPS publication FIPS 180-1, "Secure Hash Standard," April 17, 1995 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. MAC Calculation Procedure

3	Procedure name:	
4	ehmacsha	
5	Inputs from calling process:	
6	key_length	integer
7	key	key_length bits
8	message	bit string
9	MAC_length	integer
10		
11	Inputs from internal stored data:	
12	None.	
13	Outputs to calling process:	
14	MAC	8*MAC_length bits
15	Outputs to internal stored data:	
16	None.	
17		

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 IK 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 IK to create a 64 bytes string.
3. XOR (bitwise exclusive-OR) the 64 bytes string computed in step 2 with ipad defined in step 1.
4. Apply SHA-1 compression function (see 2.1.1) to the 64-bytes data string computed in step 3 loaded in the function payload, while chaining variable input is loaded with the standard IV (Initial Vector) defined in FIPS-180-1.
5. Store the result of SHA-1 compression conducted in step 4 as the intermediary key K1.

2.1.2.2. UIM-Present MAC (UMAC) Generation Procedure

Procedure name:	UMAC_Generation	
Inputs from calling process:	MAC_length	integer
	MAC	8*MAC_length bits
Inputs from internal stored data:	SHA1_INIT_UAK	160 bits
Outputs to calling process:	UMAC	8*MAC_length bits
Outputs to internal stored data:	None.	

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 sets the SHA-1 initialization vector to SHA1_INIT_UAK, then computes the SHA-1 hash of MAC, treated as a message of length 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 part using SHA-1 based MAC. The UMAC_Generation procedure shall be performed in the UIM.¹

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.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21

The UMAC_Generation procedure proceeds as follows:

1. Assemble the bit string containing the MAC of the message as described in 2.1.2.1.
2. Append the padding bit set to '1' to the least significant end of the message.
3. Append as many '0' bits as necessary to extend the length of resulting bit string to 511 bits. Append the least significant bit of the resulting bit string with an Indicator bit set to '0'.
4. Apply the SHA-1 compression function (see 2.1.1) loaded with the key UAK XOR (bitwise exclusive-OR) with the standard IV (Initial Vector) defined in FIPS-180-1 at the chaining variable input to the bit string assembled in step 3 to compute the UMAC of the message: $UMAC = f_{UAK \oplus IV}(MAC, pad, 0)$.

A test vector for UMAC_Generation is included in 4.2.

2.2. Authentication

2.2.1. UIM Authentication

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

Procedure *UMAC_Generation* (see 2.1.2.2) computes the SHA-1 hash of a MAC (typically keyed with IK), keyed with UAK.

2.2.2. One-Way Roaming to 2G systems

2.2.2.1. GSM Triplet Generation from SSD

Procedure name:

fh

Inputs from calling process:

Shared Secret Data (SSD):

SSD_A 64 bits

SSD_B 64 bits

Random Number (RAND) 128 bits

Type identifier (fh) 8 bits

Family Key (Fmk) 32 bits

Inputs from internal stored data:

None.

Outputs to calling process:

GSM Triplet:

RAND 128 bits

SRES 32 bits

Kc 64 bits

Outputs to internal stored data:

None.

The function fh can be used to generate a GSM authentication triplet to support one-way roaming from ANS-41 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 ANS-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 with known constants as follows:
Load the IV with the standard SHA IV constant.

- 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 follows:
4 XOR the SSD-A into the leftmost (most significant) 64 bits of IV,
5 and XOR the SSD-B into the next 64 bits of IV. The 64-bit internal
6 counter, initialized to 0, is XORed into the (0th, 1st) words, (4th, 5th)
7 words, (8th, 9th) words, and (12th, 13th) words of the payload. Next, a
8 “type constant” fh is XORed into the 2nd word, and the family key is
9 XORed with the 3rd word of the payload. The 128-bit random
10 number is split into two parts (64 bits each). The least significant 64
11 bits are XORed with the 6th and 7th words, and the most significant
12 64 bits are XORed with the 10th and 11th words of the payload.
- 13 3. Run SHA to produce the 160-bit output.
- 14 4. The polynomial $AX + B \text{ mod } G$ is calculated, where: A and B are
15 predetermined 160-bit random numbers (treated as polynomials with
16 binary coefficients in the variable T). A and B remain constant for
17 the life of the UIM, and can be equal for all UIMs in the system. X
18 is the 160-bit output from the SHA operation, treated as a
19 polynomial with binary coefficients in the variable T. G is the
20 polynomial $T^{160} + T^5 + T^3 + T^2 + 1$. Extract the least significant 64 bits
21 and store it in the key 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 3.2.2 for a description of this algorithm in ANSI C.

31

2.2.2.2. 2G Key Generation from 3G Keys

Procedure name:

CDMA_3G_2G_Conversion

Inputs from calling process:

AKA Cipherring Key (CK) 128 bits

Inputs from internal stored data:

None.

Outputs to calling process:

PLCM 40 bits
CMEAKEY 64 bits

Outputs to internal stored data:

None.

This procedure can be used to create the CDMA private long code mask (PLCM) and the message encryption key CMEAKEY for intersystem handoff from a system using AKA to a system using older (2G) algorithms for authentication and privacy. (On the ANS-41 network, these keys are referred to as CDMA_PLCM and SMEKEY.)

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

Procedure:

1. Calculate the 160-bit SHA-1 digest of the string consisting of the 160 bits of the ASCII string "3G_2GCDMA_CONVERSION" concatenated with the 128 bits of the cipherring key CK.
2. Set the Private Long Code Mask PLCM to the 40 least significant bits of the SHA-1 output.
3. Set the CMEA Key CMEAKEY to the next 64 least significant bits of the SHA-1 output.

See 3.2.3 for a description of this algorithm in ANSI C.

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 Common Cryptographic Algorithms, Revision D.1), the key used for the Rijndael encryption algorithm should be the 64-bit CMEAKEKEY (also sent on the ANS-41 network 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 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
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

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

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

2.3.3. Enhanced Privacy Algorithm

2.3.3.1. Algorithm

The enhanced privacy algorithm is 128-bit Rijndael. Refer to 3.1 for a reference implementation of the enhanced privacy procedures using Rijndael. Test vectors for the Rijndael algorithm are provided in 4.1.

2.3.3.2. ESP_privacykey Procedure

Procedure name:
ESP_privacykey
Inputs from calling process:
key 16*8 bits
Inputs from internal stored data:
None.
Outputs to calling process:
None.
Outputs to internal stored data:
ESP_privacykeyschedule

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

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

Procedure:

Load the 128-bit ciphering key CK in a 4 by 8 array and invoke 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

Procedure name:

ESP_maskbits

Inputs from calling process:

fresh	freshsize*8 bits
freshsize	integer
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*.
- *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
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

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

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

Procedure:

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

Load buffer *b* (a 4 by 8 array) with 1 copies of *fresh* and 16 copies of the 32-bit internal counter.

Invoke `rijndaelEncrypt()` with *b*, `KEYLENGTH*8`, `BLOCKLENGTH*8`, and *rk*.

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

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

XOR the mask into buffer

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

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
18
19
20
21
22
23
24
25

Procedure name:

ESP_AES

Inputs from calling process:

key	16*8 bits
fresh	freshsize*8 bits
freshsize	integer
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

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*.

3. Reference Implementations

3.1. Privacy

3.1.1. Rijndael

As of the time of publication of this document, a reference implementation of the Rijndael algorithm is available at <<http://csrc.nist.gov/encryption/aes/rijndael/rijndael-dos-refc.zip>>. The code is reproduced below. Note that only the functions rijndaelKeySched and rijndaelEncrypt are used for CDMA enhanced privacy.

Exhibit 3-1 Header for Rijndael

```

11 /* rijndael-alg-ref.h  v2.0  August '99
12  * Reference ANSI C code
13  * authors: Paulo Barreto
14  *          Vincent Rijmen
15  */
16 #ifndef __RIJNDAEL_ALG_H
17 #define __RIJNDAEL_ALG_H
18
19 #define MAXBC      (256/32)
20 #define MAXKC      (256/32)
21 #define MAXROUNDS  14
22
23 typedef unsigned char    word8;
24 typedef unsigned short   word16;
25 typedef unsigned long    word32;
26
27
28 int rijndaelKeySched (word8 k[4][MAXKC], int keyBits, int blockBits,
29                     word8 rk[MAXROUNDS+1][4][MAXBC]);
30 int rijndaelEncrypt (word8 a[4][MAXBC], int keyBits, int blockBits,
31                     word8 rk[MAXROUNDS+1][4][MAXBC]);
32 int rijndaelEncryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
33                           word8 rk[MAXROUNDS+1][4][MAXBC], int rounds);
34 int rijndaelDecrypt (word8 a[4][MAXBC], int keyBits, int blockBits,
35                     word8 rk[MAXROUNDS+1][4][MAXBC]);
36 int rijndaelDecryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
37                           word8 rk[MAXROUNDS+1][4][MAXBC], int rounds);
38
39 #endif /* __RIJNDAEL_ALG_H */

```

Exhibit 3-2 Rijndael Box Data

```

42 /* "boxes-ref.dat" */
43
44 word8 Logtable[256] = {
45     0,  0, 25,  1, 50,  2, 26, 198, 75, 199, 27, 104, 51, 238, 223,  3,
46    100,  4, 224, 14, 52, 141, 129, 239, 76, 113,  8, 200, 248, 105, 28, 193,

```

```

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

```

```

1   84, 123, 148, 50, 166, 194, 35, 61, 238, 76, 149, 11, 66, 250, 195, 78,
2   8, 46, 161, 102, 40, 217, 36, 178, 118, 91, 162, 73, 109, 139, 209, 37,
3  114, 248, 246, 100, 134, 104, 152, 22, 212, 164, 92, 204, 93, 101, 182, 146,
4  108, 112, 72, 80, 253, 237, 185, 218, 94, 21, 70, 87, 167, 141, 157, 132,
5  144, 216, 171, 0, 140, 188, 211, 10, 247, 228, 88, 5, 184, 179, 69, 6,
6  208, 44, 30, 143, 202, 63, 15, 2, 193, 175, 189, 3, 1, 19, 138, 107,
7   58, 145, 17, 65, 79, 103, 220, 234, 151, 242, 207, 206, 240, 180, 230, 115,
8  150, 172, 116, 34, 231, 173, 53, 133, 226, 249, 55, 232, 28, 117, 223, 110,
9   71, 241, 26, 113, 29, 41, 197, 137, 111, 183, 98, 14, 170, 24, 190, 27,
10 252, 86, 62, 75, 198, 210, 121, 32, 154, 219, 192, 254, 120, 205, 90, 244,
11 31, 221, 168, 51, 136, 7, 199, 49, 177, 18, 16, 89, 39, 128, 236, 95,
12 96, 81, 127, 169, 25, 181, 74, 13, 45, 229, 122, 159, 147, 201, 156, 239,
13 160, 224, 59, 77, 174, 42, 245, 176, 200, 235, 187, 60, 131, 83, 153, 97,
14 23, 43, 4, 126, 186, 119, 214, 38, 225, 105, 20, 99, 85, 33, 12, 125,
15 };
16
17 word8 iG[4][4] = {
18 0x0e, 0x09, 0x0d, 0x0b,
19 0x0b, 0x0e, 0x09, 0x0d,
20 0x0d, 0x0b, 0x0e, 0x09,
21 0x09, 0x0d, 0x0b, 0x0e,
22 };
23
24 word32 rcon[30] = {
25 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
26 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
27 0xfa, 0xef, 0xc5, 0x91, };
28
29

```

Exhibit 3-3 Rijndael Algorithm

```

30 /* rijndael-alg-ref.c v2.0 August '99
31 * Reference ANSI C code
32 * authors: Paulo Barreto
33 *          Vincent Rijmen
34 *
35 * This code is placed in the public domain.
36 */
37
38 #include <stdio.h>
39 #include <stdlib.h>
40
41 #include "rijndael-alg-ref.h"
42
43 #define SC((BC - 4) >> 1)
44
45 #include "boxes-ref.dat"
46
47 static word8 shifts[3][4][2] = {
48 0, 0,
49 1, 3,
50 2, 2,
51 3, 1,
52
53 0, 0,
54 1, 5,
55 2, 4,
56 3, 3,

```

```

1
2     0, 0,
3     1, 7,
4     3, 5,
5     4, 4
6 };
7
8
9 word8 mul(word8 a, word8 b) {
10     /* multiply two elements of GF(2^m)
11     * needed for MixColumn and InvMixColumn
12     */
13     if (a && b) return Alogtable[(Logtable[a] + Logtable[b])%255];
14     else return 0;
15 }
16
17 void KeyAddition(word8 a[4][MAXBC], word8 rk[4][MAXBC], word8 BC) {
18     /* Exor corresponding text input and round key input bytes
19     */
20     int i, j;
21
22     for(i = 0; i < 4; i++)
23         for(j = 0; j < BC; j++) a[i][j] ^= rk[i][j];
24 }
25
26 void ShiftRow(word8 a[4][MAXBC], word8 d, word8 BC) {
27     /* Row 0 remains unchanged
28     * The other three rows are shifted a variable amount
29     */
30     word8 tmp[MAXBC];
31     int i, j;
32
33     for(i = 1; i < 4; i++) {
34         for(j = 0; j < BC; j++) tmp[j] = a[i][(j + shifts[SC][i][d]) % BC];
35         for(j = 0; j < BC; j++) a[i][j] = tmp[j];
36     }
37 }
38
39 void Substitution(word8 a[4][MAXBC], word8 box[256], word8 BC) {
40     /* Replace every byte of the input by the byte at that place
41     * in the nonlinear S-box
42     */
43     int i, j;
44
45     for(i = 0; i < 4; i++)
46         for(j = 0; j < BC; j++) a[i][j] = box[a[i][j]] ;
47 }
48
49 void MixColumn(word8 a[4][MAXBC], word8 BC) {
50     /* Mix the four bytes of every column in a linear way
51     */
52     word8 b[4][MAXBC];
53     int i, j;
54
55     for(j = 0; j < BC; j++)
56         for(i = 0; i < 4; i++)
57             b[i][j] = mul(2,a[i][j])

```

```

1         ^ mul(3,a[(i + 1) % 4][j])
2         ^ a[(i + 2) % 4][j]
3         ^ a[(i + 3) % 4][j];
4     for(i = 0; i < 4; i++)
5         for(j = 0; j < BC; j++) a[i][j] = b[i][j];
6     }
7
8     void InvMixColumn(word8 a[4][MAXBC], word8 BC) {
9         /* Mix the four bytes of every column in a linear way
10        * This is the opposite operation of Mixcolumn
11        */
12        word8 b[4][MAXBC];
13        int i, j;
14
15        for(j = 0; j < BC; j++)
16        for(i = 0; i < 4; i++)
17            b[i][j] = mul(0xe,a[i][j])
18                ^ mul(0xb,a[(i + 1) % 4][j])
19                ^ mul(0xd,a[(i + 2) % 4][j])
20                ^ mul(0x9,a[(i + 3) % 4][j]);
21        for(i = 0; i < 4; i++)
22            for(j = 0; j < BC; j++) a[i][j] = b[i][j];
23    }
24
25    int rijndaelKeySched (word8 k[4][MAXKC], int keyBits, int blockBits, word8
26    W[MAXROUNDS+1][4][MAXBC]) {
27        /* Calculate the necessary round keys
28        * The number of calculations depends on keyBits and blockBits
29        */
30        int KC, BC, ROUNDS;
31        int i, j, t, rconpointer = 0;
32        word8 tk[4][MAXKC];
33
34        switch (keyBits) {
35        case 128: KC = 4; break;
36        case 192: KC = 6; break;
37        case 256: KC = 8; break;
38        default : return (-1);
39        }
40
41        switch (blockBits) {
42        case 128: BC = 4; break;
43        case 192: BC = 6; break;
44        case 256: BC = 8; break;
45        default : return (-2);
46        }
47
48        switch (keyBits >= blockBits ? keyBits : blockBits) {
49        case 128: ROUNDS = 10; break;
50        case 192: ROUNDS = 12; break;
51        case 256: ROUNDS = 14; break;
52        default : return (-3); /* this cannot happen */
53        }
54
55
56        for(j = 0; j < KC; j++)
57            for(i = 0; i < 4; i++)

```

```

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

```

```

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

```

```

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

```

```

1      /* End with the extra key addition
2      */
3
4      KeyAddition(a,rk[0],BC);
5
6      return 0;
7  }
8
9
10 int rijndaelDecryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
11     word8 rk[MAXROUNDS+1][4][MAXBC], int rounds)
12 /* Decrypt only a certain number of rounds.
13  * Only used in the Intermediate Value Known Answer Test.
14  * Operations rearranged such that the intermediate values
15  * of decryption correspond with the intermediate values
16  * of encryption.
17  */
18 {
19     int r, BC, ROUNDS;
20
21     switch (blockBits) {
22     case 128: BC = 4; break;
23     case 192: BC = 6; break;
24     case 256: BC = 8; break;
25     default : return (-2);
26     }
27
28     switch (keyBits >= blockBits ? keyBits : blockBits) {
29     case 128: ROUNDS = 10; break;
30     case 192: ROUNDS = 12; break;
31     case 256: ROUNDS = 14; break;
32     default : return (-3); /* this cannot happen */
33     }
34
35
36     /* make number of rounds sane */
37     if (rounds > ROUNDS) rounds = ROUNDS;
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 > rounds; r--) {
50         KeyAddition(a,rk[r],BC);
51         InvMixColumn(a,BC);
52         Substitution(a,Si,BC);
53         ShiftRow(a,1,BC);
54     }
55
56     if (rounds == 0) {
57         /* End with the extra key addition

```

```

1         */
2         KeyAddition(a,rk[0],BC);
3     }
4
5     return 0;
6 }
7

```

3.1.2. Privacy Procedures

Exhibit 3-4 Header for ESP

```

10  /* "esp.h" Header for ESP. */
11  #ifndef ESP_HEADER
12  #define ESP_HEADER
13
14  #define KEYLENGTH 16 /* octets */
15
16  /* external interface declarations */
17  void ESP_privacykey(unsigned char key[KEYLENGTH]);
18  void
19  ESP_maskbits(unsigned char *fresh,
20              int freshsize,
21              unsigned char *buf,
22              unsigned long bit_offset,
23              unsigned long bit_count);
24  void
25  ESP_AES( unsigned char key[KEYLENGTH],
26          unsigned char *fresh,
27          int freshsize,
28          unsigned char *buf,
29          unsigned long bit_offset,
30          unsigned long bit_count);
31
32  #endif
33

```

Exhibit 3-5 ESP_keysched and ESP_maskbits

```

35  /* "esp.c" */
36
37  #include "esp.h"
38
39  #include "rijndael-alg-ref.h"
40  #define BLOCKLENGTH 16 /* octets */
41
42  /* internal storage */
43
44  /* ESP_privacykeyschedule consists of the array rk[[]]. */
45
46  static
47  unsigned char rk[MAXROUNDS+1][4][MAXBC]; /* rijndael Key Schedule */
48
49  /* Schedule key in internal storage. */
50
51  void
52  ESP_privacykey(unsigned char key[KEYLENGTH])

```

```

1  {
2      unsigned char k[4][MAXKC];
3      int          i;
4
5      /* reshape key for rijndael */
6      for (i = 0; i < KEYLENGTH; ++i)
7          k[i%4][i/4] = key[i];
8      rijndaelKeySched(k, KEYLENGTH*8, BLOCKLENGTH*8, rk);
9  }
10
11
12 /* encrypt/decrypt a buffer of data or output an encryption mask */
13
14 void
15 ESP_maskbits(unsigned char *fresh,
16              int freshsize,
17              unsigned char *buf,
18              unsigned long bit_offset,
19              unsigned long bit_count)
20 {
21     int          i;
22     unsigned long counter;
23     unsigned char *bptr;
24     unsigned char b[4][MAXBC], offset, mask, bit_mask, mask_size, last_mask;
25
26     if (bit_count <= 0)
27         return;
28
29     offset = (unsigned char)(bit_offset % 8);
30
31     /* point to first byte to be changed */
32     bptr = buf + (bit_offset/8);
33
34     for (counter = 0; bit_count > 0; ++counter)
35     {
36         /* initialise buffer with copies of fresh and counter */
37         for (i = 0; i < freshsize; ++i)
38             b[i%4][i/4] = fresh[i];
39         for (/* leftover i */ ; i < BLOCKLENGTH; ++i)
40             /* counter MSB first */
41             b[i%4][i/4] = (unsigned char)(counter >> ((3 - (i%4)) * 8));
42
43         /* run Rijndael */
44         rijndaelEncrypt(b, KEYLENGTH*8, BLOCKLENGTH*8, rk);
45
46         /* set up to use the first bits of the Rijndael buffer */
47         if (offset != 0)
48         {
49             /* set last mask octet to zero */
50             last_mask = 0;
51
52             /* set first mask and its size */
53             mask_size = 8 - offset;
54             bit_mask = 0xff >> offset;
55         }
56         else
57         {

```

```

1      mask_size = 8;
2      bit_mask = 0xff;
3  }
4
5      /* adjust for short remaining bit count */
6      if (bit_count < mask_size)
7      {
8          bit_mask &= 0xff << (mask_size - bit_count);
9          mask_size = (unsigned char)bit_count;
10     }
11
12     /* XOR mask into buffer */
13     for (i = 0; i < BLOCKLENGTH; ++i)
14     {
15         if (offset != 0)
16         {
17             mask = last_mask << (8 - offset);
18             last_mask = b[i%4][i/4];
19             mask |= last_mask >> offset;
20         }
21         else
22             mask = b[i%4][i/4];
23         *bptr++ ^= mask & bit_mask;
24         bit_count -= mask_size;
25         if (bit_count <= 0)
26             return;
27
28         /* set next mask and its size */
29         if (bit_count > 7)
30         {
31             mask_size = 8;
32             bit_mask = 0xff;
33         }
34         else
35         {
36             mask_size = (unsigned char)bit_count;
37             bit_mask = 0xff << (8 - mask_size);
38         }
39     }
40
41     /* use the last bits in the Rijndael buffer, if any */
42     if (offset != 0)
43     {
44         *bptr ^= (last_mask << (8 - offset)) & bit_mask;
45         if (mask_size > offset)
46             mask_size = offset;
47         bit_count -= mask_size;
48         if (bit_count <= 0)
49             return;
50     }
51 }
52 }
53
54 void
55 ESP_AES( unsigned char key[KEYLENGTH],
56          unsigned char *fresh,
57          int freshsize,

```

```

1         unsigned char *buf,
2         unsigned long bit_offset,
3         unsigned long bit_count)
4     {
5         ESP_privacykey(key);
6         ESP_maskbits(fresh, freshsize, buf, bit_offset, bit_count);
7     }
8

```

3.1.3. KeyStrengthRedAlg Function

Exhibit 3-6 KeyStrengthRedAlg Function Header

```

11 #ifndef KEYSRA
12 #define KEYSRA
13
14 void KeyStrengthRedAlg(
15     /* input */ int KeyLength, /* in octets */
16     /* input */ unsigned char *OriginalKey,
17     /* input */ int SaltLength, /* in octets */
18     /* input */ unsigned char *Salt,
19     /* input */ int KeyEntropy, /* in octets */
20     /* output */ unsigned char *RedStrengthKey
21     /* KeyLength octets in length */
22 );
23 #endif
24
25

```

Exhibit 3-7 KeyStrengthRedAlg Function Code

```

26 /* Key Strength Reduction Algorithm: "KeySRA.c"*/
27
28 #include "sha.h"
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     /* Key_Length octets in length */
38 )
39 {
40     int i;
41     SHA_INFO s;
42     unsigned char intermediateKey[DIGEST_LENGTH];
43
44     if (KeyLength > DIGEST_LENGTH)
45         KeyLength = DIGEST_LENGTH;
46
47     if (KeyEntropy > KeyLength)
48         KeyEntropy = KeyLength;
49
50     shaInitial(&s);
51     shaUpdate(&s, OriginalKey, 0, KeyLength*8);
52     shaUpdate(&s, Salt, 0, SaltLength*8);

```

```

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

```

17 **3.2. Authentication**

18 **3.2.1. SHA-1**

19 **Exhibit 3-8 SHA-1 Header**

```

20 /* "sha.h" */
21
22 #ifndef SHA_H
23 #define SHA_H
24
25 /* header for SHA and related procedures */
26 #define WORD unsigned long
27 #define DIGEST_LENGTH 20
28 typedef struct {
29     unsigned char digest[DIGEST_LENGTH]; /* Message digest */
30     WORD count[2]; /* count of bits */
31     WORD data[16]; /* data buffer */
32 }
33     SHA_INFO;
34
35 void shaInitial(SHA_INFO *sha_info);
36 void shaUpdate(SHA_INFO *sha_info,
37               unsigned char *buffer,
38               unsigned long offset,
39               unsigned long count);
40 void shaFinal(SHA_INFO *sha_info);
41
42 #endif
43

```

44 **Exhibit 3-9 SHA-1 Code**

```

45 /* "sha.c" */
46
47 #include "sha.h"
48
49 static unsigned long A, B, C, D, E;
50

```

```

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];

```

```

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

```

```

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

```

```

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

```

```

1
2     /* put in the bit length */
3
4     sha_info->data[14] = sha_info->count[0];
5     sha_info->data[15] = sha_info->count[1];
6
7     /* update the digest */
8
9     shaHash(sha_info);
10  }
11

```

3.2.2. GSM Triplet Generation Function fh

Exhibit 3-10 Function fh Header

```

14  /* gsmlway.h */
15
16  #ifndef GSM1WAY
17  #define GSM1WAY
18
19  #include "sha.h"
20
21  typedef unsigned char      uchar;
22  typedef unsigned short    word16;
23  typedef unsigned long int word32;
24
25  #define L_KEY      16      /*128      bits 3G TS 33.105.v.3.0(2000-03)*/
26  #define L_RAND     16      /*128      bits 3G TS 33.105.v.3.0(2000-03)*/
27  #define L_SQN      6       /*48       bits 3G TS 33.105.v.3.0(2000-03)*/
28  #define L_FMK      4       /*32       bits */
29  #define L_CK       16      /*128      bits 3G TS 33.105.v.3.0(2000-03)*/
30
31  typedef struct {
32      unsigned char RAND[16];
33      unsigned char SRES[4];
34      unsigned char Kc[8];
35  } GSM_triplet_type;
36
37  void
38  fh(uchar SSD_A[],uchar SSD_B[],uchar RAND[],uchar fhi,
39     uchar Fmk[],GSM_triplet_type *t);
40
41  #endif
42

```

Exhibit 3-11 Function fh Code

```

44  /* gsmlway.c */
45
46  #include "sha.h"
47  #include "gsmlway.h"
48
49  static uchar counter[8]={0};
50
51  static uchar G[20] = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
52                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

```

```

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

```

```

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

```

```

1      sha_info at the start of this loop. */
2      shaInitial(&sha_info);
3      for (i = 0; i < 8; i++)
4      {
5          sha_info.digest[i] ^= SSD_A[i];
6          sha_info.digest[i+8] ^= SSD_B[i];
7      }
8
9      for (i = 0; i < 64; i++)
10         temp[i] = 0x5c;
11
12     for (i = 0; i < 4; i++)
13         temp[i+12] ^= Fmk[i];
14     for (i = 0; i < 16; i++)
15         temp[i+24] ^= RAND[i];
16     temp[3] ^= j;
17     temp[11] ^= fhi;
18     temp[19] ^= j;
19     temp[35] ^= j;
20     temp[51] ^= j;
21
22     shaUpdate(&sha_info,temp,0,512);
23
24     whiten(sha_info.digest);
25
26     for (i=0;i<8;i++)
27         buf[8*j+i] = sha_info.digest[i];
28     }
29     for (i = 0; i < 16; i++)
30         t->RAND[i] = RAND[i];
31
32     for (i=0;i<8;i++)
33         t->Kc[i] = buf[i];
34
35     for (i=0;i<4;i++)
36         t->SRES[i] = buf[i+8];
37 }
38

```

3.2.3. CDMA_3G_2G_Conversion Function

Exhibit 3-12 CDMA_3G_2G_Conversion Function Header

```

41 /* twoglway.h */
42
43 #ifndef TWOGLWAY
44 #define TWOGLWAY
45
46 typedef unsigned char    uchar;
47 typedef unsigned short   word16;
48 typedef unsigned long int word32;
49
50 void
51 CDMA_3G_2G_Conversion(uchar CK[],uchar PLCM[],uchar CMEAKEY[]);
52
53 #endif
54

```

Exhibit 3-13 CDMA_3G_2G_Conversion Function Code

```

1
2  /* twoglway.c */
3
4  #include <string.h>
5  #include "sha.h"
6  #include "twoglway.h"
7
8  /* This function performs generation of 2G privacy and
9   * encryption keys from CK.
10   *
11   */
12
13 void
14 CDMA_3G_2G_Conversion(uchar CK[],uchar PLCM[],uchar CMEAKEY[])
15 {
16     SHA_INFO sha_info;
17     uchar *sha_data = "3G_2GCDMA_conversion";
18     int i;
19
20     shaInitial(&sha_info);
21     shaUpdate(&sha_info,sha_data,0,8*strlen(sha_data));
22     shaUpdate(&sha_info,CK,0,64);
23     shaFinal(&sha_info);
24
25     for (i = 0; i < 5; i++)
26         PLCM[i] = sha_info.digest[i];
27     for (i = 0; i < 8; i++)
28         CMEAKEY[i] = sha_info.digest[i+5];
29 }
30

```

3.3. EHMACHA-1

Exhibit 3-14 EHMACHA Header

```

33 /* ehmacsha.h */
34
35 #ifndef EHMACHA_H
36 #define EHMACHA_H
37 #include "sha.h"
38
39 #define HMAC_IPAD 0x36
40 #define HMAC_OPAD 0x5c
41
42 void ehmacsha_keygen(SHA_INFO *sha_info,
43                     unsigned char *key,
44                     int l_key,
45                     unsigned char pad);
46 void ehmacsha_f(SHA_INFO *sha_info_i,
47                SHA_INFO *sha_info_o,
48                unsigned char *message,
49                unsigned long message_offset,
50                unsigned long message_length,
51                unsigned char *hmac,
52                int l_hmac);

```

```

1 void ehmacsha(unsigned char Key[],
2             int l_key,
3             unsigned char *message,
4             unsigned long message_offset,
5             unsigned long message_length,
6             unsigned char *hmac,
7             int l_hmac);
8
9 void ehmacsha_s(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 void ehmacsha_m(SHA_INFO *sha_info_i, 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 #endif
23
24 Exhibit 3-15 EHMACH Code

```

```

25 /* ehmacsha.c - reference implementation of Enhanced HMAC-SHA-1 */
26
27 #include "ehmacsha.h"
28
29 unsigned char ehmac_pad[] = { 0x80,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
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                             0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
33 unsigned char multiple_block_indicator = { 0x80 };
34
35 /* compute shaHash(key xor pad), the first step in the computation of
36 the inner and outer parts of HMAC. This function must be called twice
37 before calling hmacsha(). The first step sets pad equal to HMAC_IPAD
38 and the third step sets pad equal to HMAC_OPAD. These steps can be
39 precomputed and the sha_info from each step saved for future HMAC
40 computations using the same key. */
41
42 void ehmacsha_keygen(SHA_INFO *sha_info,
43                    unsigned char *key,
44                    int l_key,
45                    unsigned char pad)
46 {
47     unsigned char buf[64],*kptr;
48     int i,l;
49
50     if (l_key > 64)
51     {
52         shaInitial(sha_info);
53         shaUpdate(sha_info,key,0,8*l_key);
54         shaFinal(sha_info);
55         kptr = sha_info->digest;
56         l = 20;

```

```

1     }
2     else
3     {
4         kptr = key;
5         l = l_key;
6     }
7
8     for (i = 0; i < 64; i++)
9     {
10        buf[i] = pad;
11        if (i < l)
12            buf[i] ^= kptr[i];
13    }
14
15    shaInitial(sha_info);
16    shaUpdate(sha_info,buf,0,512);
17 }
18
19 /* Compute EHMAL (typically using IK) for the single block case.
20 hmacsha_keygen() must be called to initialize sha_info_o before
21 calling this function. */
22
23
24 void ehmacsha_s(SHA_INFO *sha_info_o,
25                unsigned char *message,
26                unsigned long message_offset,
27                unsigned long message_length,
28                unsigned char *ehmac,
29                int l_ehmac)
30 {
31     SHA_INFO sha_info;
32     int i;
33
34     sha_info = *sha_info_o;
35
36     shaUpdate(&sha_info,message,message_offset,message_length);
37
38     /* add mandatory 1 and as many zeroes as necessary to fill 512 bits*/
39     shaUpdate(&sha_info, ehmac_pad, 0, 512 - message_length);
40
41     for (i = 0; i < l_ehmac; i++)
42         ehmac[i] = sha_info.digest[i];
43 }
44
45 /* Compute EHMAL (typically using IK) for the multiple block case.
46 hmacsha_keygen() must be called to initialize sha_info_i & sha_info_o before
47 calling this function. */
48
49 static
50 void ehmacsha_m(SHA_INFO *sha_info_i, 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;unsigned char digest[20];

```

```

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

```

```

1     ehmacsha_keygen(&sha_info_o,Key,l_key,HMAC_OPAD);
2     ehmacsha_m(&sha_info_i,&sha_info_o,message,message_offset,
3               message_length,ehmac,l_ehmac);}
4 }

```

```

5
6

```

Exhibit 3-16 UMAC_Generation Code

```

7 /* umac.c */
8
9 #include "ehmacsha.h"
10
11 void UMAC_Generation(unsigned char UAK[],
12                    int L_UAK,
13                    unsigned char MAC[],
14                    int l_mac,
15                    unsigned char UMAC[])
16 {
17     SHA_INFO sha_info;
18     int i;
19
20     if (l_mac < 1)
21         return;
22     if (l_mac > 64)
23         l_mac = 64;
24
25     /* load the UAK into sha_info. Note that this can be done once when
26        UAK is generated, and the sha_info copied from storage, to save
27        time. */
28
29     shaInitial(&sha_info);
30     for (i = 0; i < L_UAK; i++)
31         sha_info.digest[i] ^= UAK[i];
32
33     ehmacsha_s(&sha_info,MAC,0,8*l_mac,UMAC,l_mac);
34
35 }
36

```

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 Rijndael 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 Rijndael 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");

```

```

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

```

```

1
2     ESP_privacykey(tkey);
3     ESP_maskbits(fresh,8,buf,5,8*40);
4
5     for (i = 0; i < 16; i++)
6         printf("%02x ",buf[i]);
7     printf("\n");
8     for (i = 16; i < 32; i++)
9         printf("%02x ",buf[i]);
10    printf("\n");
11    for (i = 32; i < 41; i++)
12        printf("%02x ",buf[i]);
13    printf("\n");
14
15    pause();
16
17    /* bit_offset = 3; bit_count = 8*32+3 */
18
19    printf("bit_offset = %d; bit_count = %d\n\n",3,8*32+3);
20
21    for (i = 0; i < 41; i++)
22        buf[i] = 0;
23
24    ESP_privacykey(tkey);
25    ESP_maskbits(fresh,8,buf,3,8*32+3);
26
27    for (i = 0; i < 16; i++)
28        printf("%02x ",buf[i]);
29    printf("\n");
30    for (i = 16; i < 32; i++)
31        printf("%02x ",buf[i]);
32    printf("\n");
33    for (i = 32; i < 41; i++)
34        printf("%02x ",buf[i]);
35    printf("\n");
36
37    pause();
38 }
39

```

4.2. Test Vectors for EHMAL-SHA-1

4.2.1. Test Program Output

```

42
43 test vector for EHMAL
44 input section
45 IK[16 bytes]:  c1 43 65 25 fa 60 7f 17 92 fc a8 9f b2 a7 bc 4a
46 UAK[16 bytes]:  55 01 c0 20 86 9b 8f ef 7a 33 bb 12 a0 d0 2e 63
47 msg[66 bytes]:
48 "abcdbcdecdefdefgefghfghighijhijkiijkljklmklmnlmnomnopnopqopqrpqrsqr"
49
50 output section
51 ehmac ( 12-bit msg):  f3 61 35 21 91 51 51 5d 4e 5d 57 11 b4 79 62 dd 79 c0 05 2b
52 ehmac (510-bit msg):  a0 ea ed 4b 19 9e a9 8c f4 bc 92 89 89 43 b5 e2 28 aa b7 5a
53 ehmac (511-bit msg):  02 81 50 67 ac a0 29 77 ae 2e 73 13 fd d6 f9 d0 55 be 37 fa

```

```

1  ehmac (520-bit msg):  70 3b de d1 34 3d 73 e9 80 e7 6a 22 9b c3 74 cd 43 bb c2 e6
2  umac (520-bit msg):  c1 04 54 af 0b 8f 6b 6b 00 b4 32 54 c2 8a 5a 36 37 90 ee 16
3

```

4.2.2. Test Program

```

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

```

```

1     printf("%02x ", umac[i]);
2     printf("\n");
3     ehmacsha(IK, 16, ehmac_test, 0, 510, umac, 20);
4     printf("ehmac (510-bit msg): ");
5     for (i=0; i<20; i++)
6         printf("%02x ", umac[i]);
7     printf("\n");
8
9     /* run EHMACHA-SHA on a long message (511 bits long) */
10    /* and verify that mode transition takes place */
11    ehmacsha(IK, 16, ehmac_test, 0, 511, umac, 20);
12    printf("ehmac (511-bit msg): ");
13    for (i=0; i<20; i++)
14        printf("%02x ", umac[i]);
15    printf("\n");
16
17    /* run EHMACHA on longer-than-buffer msg */
18    ehmacsha(IK, 16, ehmac_test, 0, 520, umac, 20);
19    printf("ehmac (520-bit msg): ");
20    for (i=0; i<20; i++)
21        printf("%02x ", umac[i]);
22    printf("\n");
23
24    /* create the UMAC for this result */
25    UMAC_Generation(UAK,L_UAK,umac,20,umac);
26    printf("umac (520-bit msg): ");
27    for (i=0; i<20; i++)
28        printf("%02x ", umac[i]);
29    printf("\n");
30    printf("\n");
31
32    pause();
33
34 }
35

```

36 4.3. Test Vectors for One-Way Roaming to 2G Systems

37 4.3.1. Test Program Output

```

38 test vector for fh
39 input section
40 SSD_A is:  ad 1b 5a 15 9b e8 6b 2c
41 SSD_B is:  a6 6c 7a e4 0b ba 9b 9d
42 RAND is:   4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
43 fih is:    60
44 Fmk is:   41 48 41 47
45
46 output section
47 fh Kc:    1b 08 ad 36 44 ba 2a 85
48 fh SRES:  92 06 4a d2
49
50
51 test vector for 3G-2G conversion
52 input section
53 CK is:   6e fd d8 32 f6 ff d4 dc a8 4a 54 96 fa 6e 29 93

```

```

1
2 output section
3 2G PLCM: 52 16 ad b2 9e
4 2G CMEAKEY: 9d fd d1 45 a9 fe 45 31
5

```

4.3.2. Test Program

```

7 /* test_fh and 3G to 2G conversion */
8
9 #include <stdio.h>
10 #include "gsmlway.h"
11 #include "twoglway.h"
12
13 void pause(void)
14 {
15     printf("Press any key to continue\n");
16     getchar();
17 }
18
19 void main()
20 {
21     uchar K[]={
22         0xad,0x1b,0x5a,0x15,0x9b,0xe8,0x6b,0x2c,
23         0xa6,0x6c,0x7a,0xe4,0x0b,0xba,0x9b,0x9d };
24
25     uchar seed[]={
26         0xb0,0xab,0xb9,0x9d,0x6a,0xc6,0xa7,0x4e,
27         0xb9,0x8e,0xb6,0xc2,0xda,0xb1,0xa5,0x51 };
28
29     uchar Fmk[L_FMK] = { 'A', 'H', 'A', 'G' };
30     uchar RAND[L_RAND] = {
31         0x4b, 0x05, 0x2b, 0x20, 0xe2, 0xa0, 0x6c, 0x8f,
32         0xf7, 0x00, 0xda, 0x51, 0x2b, 0x4e, 0x11, 0x1e };
33     uchar CK[L_CK] = {
34         0x6e, 0xfd, 0xd8, 0x32, 0xf6, 0xff, 0xd4, 0xdc,
35         0xa8, 0x4a, 0x54, 0x96, 0xfa, 0x6e, 0x29, 0x93 };
36
37     GSM_triplet_type triplet;
38     uchar CMEAKEY[8];
39     uchar PLCM[5];
40
41     uchar SQN[L_SQN]={0x00,0x00,0x00,0x00,0x00,0x01};
42     uchar fih;
43
44     int i;
45
46     fih = 0x60;
47
48     printf("\n");
49     printf("\n");
50     printf("test vector for fh\n");
51     printf("input section\n");
52     printf("SSD_A is: ");
53     for(i=0;i<L_KEY/2;i++)
54         printf("%02x ",K[i]);
55     printf("\n");

```

```

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

```