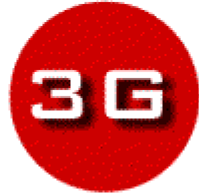
*3GPP2 S.S0055*

*Version 1.0*

*Version Date: 21 January 2002*



3RD GENERATION
PARTNERSHIP
PROJECT 2
"3GPP2"

# ***Enhanced Cryptographic Algorithms***


*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.*

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* |
| ***0.1*** | *First draft* | *01-21-02* |

9

# Table of Contents

1. INTRODUCTION 1

1.1. Notations 1

1.2. Definitions 1

2. PROCEDURES 2

2.1. Enhanced Hash Algorithm 2
2.1.1. SHA-1 2

2.2. Authentication and Key Agreement 2
2.2.1. AKA 2
2.2.2. SHA-Based Functions for AKA 2
2.2.2.1. Constants 3
2.2.2.2. Random Number (RAND) Generation Procedure *f0* 4
2.2.2.3. Message Authentication (MACA) Generation Procedure *f1* 6
2.2.2.4. Resynchronization Message Authentication (MACS) Generation Procedure *f1** 7
2.2.2.5. Message Authentication (RES & XRES) Generation Procedure *f2* 8
2.2.2.6. Ciphering Key (CK) Generation Procedure *f3* 9
2.2.2.7. Integrity Key (IK) Generation Procedures *f4* 11
2.2.2.8. Anonymity Key (AK) Generation Procedure *f5* 12
2.2.2.9. Resynchronization Anonymity Key (AKS) Generation Procedure *f5** 13

2.3. Enhanced Voice and Data Privacy 14
2.3.1. TDMA (TIA-136) 14
2.3.2. CDMA (TIA/EIA/IS-2000) 14
2.3.2.1. Encryption Key Generation 14
2.3.2.2. Enhanced Privacy Algorithm 14
2.3.2.2.1. Algorithm 14
2.3.2.2.2. ESP_privacykey Procedure 15
2.3.2.2.3. ESP_maskbits Procedure 16
2.3.2.2.4. ESP_AES Procedure 17

3. REFERENCE IMPLEMENTATIONS 18

3.1. CDMA Enhanced Privacy 18
3.1.1. Rijndael 18
3.1.2. ESP Procedures 25

3.2. SHA-Based AKA Functions 28
3.2.1. SHA-1 28
3.2.2. AKA Functions f0-f5 33

4. TEST VECTORS 42

4.1. CDMA Enhanced Privacy 42
4.1.1. Test Program Output 42
4.1.2. Test Program 42

1 **4.2. SHA-Based Functions for AKA 44**
2 4.2.1. Test Program Output 44
3 4.2.2. Test Program 46
4

# List of Exhibits

EXHIBIT 2-1. PSEUDO RANDOM GENERATOR. .......... 5
EXHIBIT 2-2. KEY SCHEDULER .......... 10
EXHIBIT 3-1 HEADER FOR RIJNDAEL .......... 18
EXHIBIT 3-2 RIJNDAEL BOX DATA .......... 18
EXHIBIT 3-3 RIJNDAEL ALGORITHM .......... 20
EXHIBIT 3-4 HEADER FOR ESP .......... 25
EXHIBIT 3-5 ESP_KEYSCHED AND ESP_MASKBITS .......... 26
EXHIBIT 3-6 SHA-1 HEADER .......... 28
EXHIBIT 3-7 SHA-1 CODE .......... 28
EXHIBIT 3-8 AKA FUNCTION HEADER .......... 33
EXHIBIT 3-9 AKA FUNCTION CODE .......... 34
EXHIBIT 4-1 RIJNDAEL TEST OUTPUT .......... 42
EXHIBIT 4-2 RIJNDAEL TEST PROGRAM .......... 42
EXHIBIT 4-3 AKA FUNCTION TEST OUTPUT .......... 44
EXHIBIT 4-4 AKA FUNCTION TEST PROGRAM .......... 46

# 1. Introduction

This document describes detailed cryptographic procedures for wireless system applications. These procedures are used to perform the security services of mutual authentication between mobile stations and base stations, subscriber message encryption, and key agreement within wireless equipment.

## 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.

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.

# 2. Procedures

## 2.1. Enhanced 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.

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

## 2.2. Authentication and Key Agreement

### 2.2.1. AKA

The procedures for Authentication and Key Agreement (AKA) shall be as specified in the following documents. These documents are available from the Alliance for Telecommunication Industry Solutions (ATIS) http://www.atis.org.

T1TRQ3GPP 33.102-350, “3G Security – Security Architecture,” July, 2000.

T1TRQ3GPP 33.103-330, “3G Security – Integration Guidelines,” July, 2000.

T1TRQ3GPP 33.105-340, “Cryptographic Algorithm Requirements,” July, 2000.

### 2.2.2. SHA-Based Functions for AKA

This section provides the interface to a reference implementation of the AKA functions f0-f5 using the compression function of SHA-1. Refer to 3.2.2 for the reference implementation of f0-f5. The use of the reference functions as defined herein is recommended but not required, since these AKA functions are implemented only in the home system Authentication Center and in the UIM. Any set of functions that are equivalent in cryptographic strength and compatible with regard to input and output sizes can be used.

Test vectors for these functions are given in 4.1.

#### 2.2.2.1. Constants

The following constants are used in these AKA functions: Fmk, A, B, f0, f1, f1*, f2, f3, f4, f5 and f5*. Constants A and B are 160 bits, constants f0, f1, f1*, f2, f3, f4, f5 and f5* are 8 bits.

Fmk is a “family key”, with the fixed value 0x41484147. Use of different values for this constant would allow other “versions” or “flavors” of the basic AKA algorithms. In a mobile station with a non-removable UIM, the use of an Fmk value other than the standard value can prevent re-provisioning of that mobile station except with the original service provider.

A and B are as the least significant bits of the first 320 digits of the RAND Corporation book of random numbers, online at http://www.rand.org/publications/classics/randomdigits. Bit 1 and bit 161 are the MSB bits of A and B respectively. A and B are specified as follows:

```
A = 0x9DE9C9C8EFD5781148231401901F2D493F4C6365
B = 0x75EFD15C4B8F8F514EF3BCC3794A765E7EEC45E0
```

The values of the 8-bit type identifiers for the functions f0, f1, f1*, f2, f3, f4, f5 and f5* are specified as follows:

f0 = 0x41
f1 = 0x42
f1* = 0x43
f2 = 0x44
f3 = 0x45
f4 = 0x46
f5 = 0x47
f5* = 0x48

#### 2.2.2.2. Random Number (RAND) Generation Procedure *f0*

| Procedure name: | |
|---|---|
| f0 | |
| Inputs from calling process: | |
| Random Secret Seed (K) | 128 bits |
| Type identifier (f0) | 8 bits |
| Family Key (Fmk) | 32 bits |
| Inputs from internal stored data: | |
| Counter (counter) | 64 bits |
| Outputs to calling process: | |
| RAND | 64 bits |
| Outputs to internal stored data: | |
| None. | |

The function f0 can be used to generate RAND values to be used in Authentication Vectors. This procedure is only used by the Authentication Center.

The function f0 is a pseudo random generator algorithm that is provably as hard as SHA-1. The function is presented with the random secret Seed, which is chosen by the operator, as well as a Counter parameter, which is initialized to 0 at the Authentication Center (AC) once, and is incremented every time the function is called. The procedure returns 64 pseudo random bits every time it is invoked. The calling process is responsible for incrementing the counter and repeatedly calling the procedure in order to generate the required number of pseudo random bits.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times

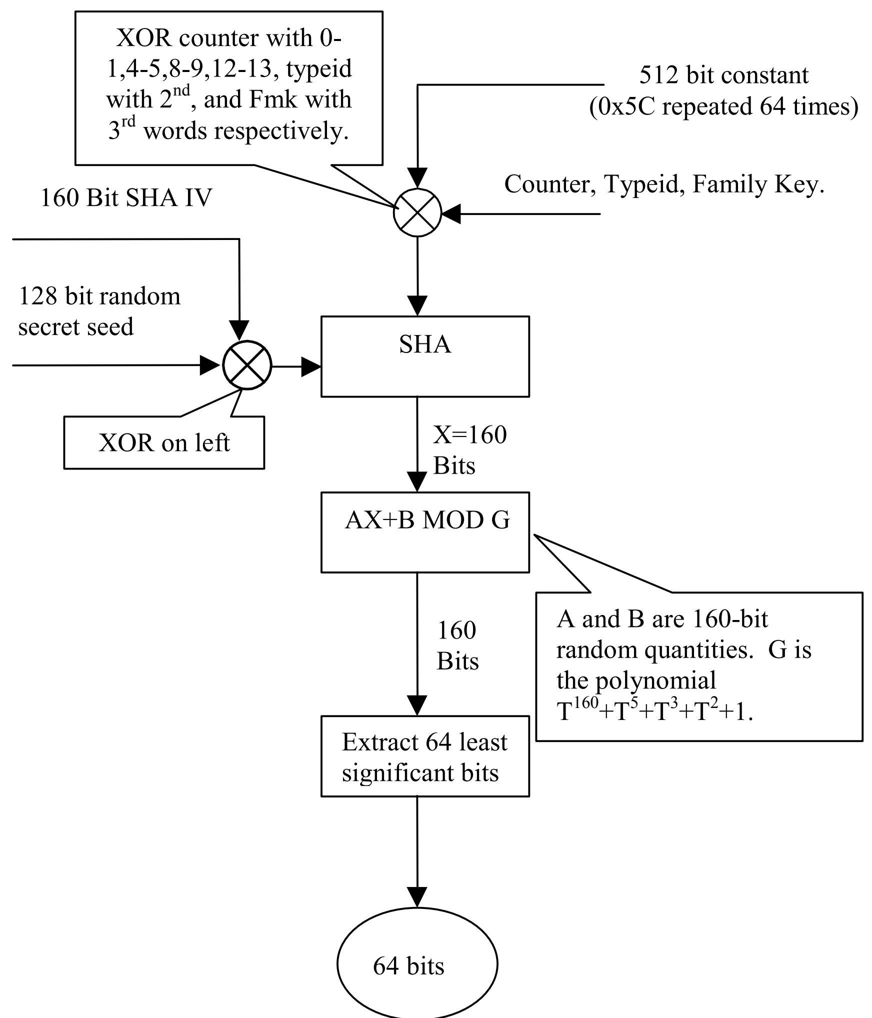
2. Load seed and counter values as follows:
   XOR seed into the leftmost (most significant) 128 bits of IV. The 64-bit counter value is XORed into the ($0^{th}$, $1^{st}$) words, ($4^{th}$, $5^{th}$) words, ($8^{th}$, $9^{th}$) words, and ($12^{th}$, $13^{th}$) words. $0^{th}$ is the least significant word and $15^{th}$ is the most significant. Next, a “type constant” (unique to function f0) is XORed into the $2^{nd}$ word, and the “family key” is XORed into the $3^{rd}$ word.

3. Run SHA to produce a 160-bit output.

4. The polynomial (AX + B mod G) is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T). A and B remain constant for the life of the UIM, and can be equal for all UIMs in the system.
   X is the 160-bit output from the SHA operation, treated as a polynomial with binary coefficients in the variable T.
   G is the polynomial $T^{160}+T^5+T^3+T^2+1$.
5. The Least Significant 64 bits of the result are returned and stored in a buffer.
6. The next time the function is invoked, increment the counter value by 1 and repeat steps 1 through 5.
7. Repeat procedure as many times as needed to obtain the required multiple of 64 bits. (When fewer than 64 bits are needed, the least significant bits should be used and the rest discarded.)

This procedure is illustrated in Exhibit 2-1.

**Exhibit 2-1. Pseudo Random Generator.**



#### 2.2.2.3. Message Authentication (MACA) Generation Procedure *f1*

Procedure name:

f1

Inputs from calling process:

| | |
|---|---|
| Subscriber Authentication Key (K) | 128 bits |
| Random Number (RAND) | 128 bits |
| Family Key (Fmk) | 32 bits |
| Type Identifier (f1) | 8 bits |
| SQN (SQN) | 48 bits |
| AMF(AMF) | 16 bits |

Inputs from internal stored data:

None.

Outputs to calling process:

| | |
|---|---|
| MACA(MACA) | 64 bits |

Outputs to internal stored data:

None.

The function f1 is used to compute the MAC that authenticates an Authentication Vector to the UIM.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times

2. Load the subscriber authentication key and counter value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. A "type constant" (unique to function f1) is XORed into the $2^{nd}$ word. The 32-bit family key is XORed into the $3^{rd}$ word, 128-bit RAND is XORed into words $4\text{-}7^{th}$. The 48-bit SQN is XORed into the $8^{th}$ and $9^{th}$ words, and the AMF is XORed into the $10^{th}$ word.

3. Run SHA to produce a 160-bit output.

4. The polynomial (AX + B mod G) is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T). A and B remain constant for the life of the UIM, and can be equal for all UIMs in the system.
   X is the 160-bit output from the SHA operation, treated as a polynomial with binary coefficients in the variable T.
   G is the polynomial $T^{160}+T^5+T^3+T^2+1$.

5. Only the 64 least significant bits are used.

#### 2.2.2.4. Resynchronization Message Authentication (MACS) Generation Procedure *f1**

Procedure name:

f1*

Inputs from calling process:

| | |
|---|---|
| Subscriber Authentication Key (K) | 128 bits |
| Random Number (RAND) | 128 bits |
| Family Key (Fmk) | 32 bits |
| Type Identifier (f1*) | 8 bits |
| SQN (SQN) | 48 bits |
| AMF(AMF) | 16 bits |

Inputs from internal stored data:

None.

Outputs to calling process:

None.

Outputs to internal stored data:

| | |
|---|---|
| MACS(MACS) | 64 bits |

The function f1* is used to compute the MAC that authenticates a resynchronization message to the Authentication Center.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times

2. Load the subscriber authentication key and counter value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. A “type constant” (unique to function f1*) is XORed into the 2nd word. The 32-bit family key is XORed into the 3rd word, 128-bit RAND is XORed into words 4-7th. The 48-bit SQN is XORed into the 8th and 9th words, and the AMF is XORed into the 10th word.

3. Run SHA to produce a 160-bit output.

4. The polynomial (AX + B mod G) is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T). A and B remain constant for the life of the UIM, and can be equal for all UIMs in the system.

X is the 160-bit output from the SHA operation, treated as a polynomial with binary coefficients in the variable T.

G is the polynomial $T^{160}+T^5+T^3+T^2+1$.

5. Only the 64 least significant bits are used.

#### 2.2.2.5. Message Authentication (RES & XRES) Generation Procedure *f2*

Procedure name:

f2

Inputs from calling process:

| | |
|---|---|
| Subscriber Authentication Key (K) | 128 bits |
| Random Number (RAND) | 32 bits |
| Type Identifier (f2) | 8 bits |

Inputs from internal stored data:

None.

Outputs to calling process:

| | |
|---|---|
| Response(RES) | 128 bits |

Outputs to internal stored data:

None.

The function f2 is used to compute the challenge response returned from the UIM when an Authentication Vector is processed.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times

2. Load the subscriber authentication key and counter value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. A "type constant" (unique to function f2) is XORed into the 2nd word. The 32-bit family key is XORed into the 3rd word, 128-bit RAND is XORed into words 4-7th.

3. Run SHA to produce a 160-bit output.

4. The polynomial (AX + B mod G) is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T). A and B remain constant for the life of the UIM, and can be equal for all UIMs in the system.
   X is the 160-bit output from the SHA operation, treated as a

polynomial with binary coefficients in the variable T.
G is the polynomial $T^{160}+T^5+T^3+T^2+1$.

5. Only the 128 least significant bits are used.

#### 2.2.2.6. Ciphering Key (CK) Generation Procedure *f3*

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:

Cipher Key (CK) 128 bits

Outputs to internal stored data:

None.

The function f3 is a pseudo random function used to generate a ciphering key. The output can be used as the key CK in an AKA Authentication Vector, or for other purposes.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times

2. Load the subscriber authentication key and index value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. The 64-bit index value, initialized to 0, is XORed into the ($0^{th}$, $1^{st}$) words, ($4^{th}$, $5^{th}$) words, ($8^{th}$, $9^{th}$) words, and ($12^{th}$, $13^{th}$) words. $0^{th}$ is the least significant word and $15^{th}$ is the most significant. Next, a "type constant" (unique to function f3) is XORed into the $2^{nd}$ word, and the family key is XORed with the $3^{rd}$ word. The 128-bit random number is split into two parts (64 bits each). The least significant 64 bits are XORed with the $6^{th}$ and $7^{th}$ words, and the most significant 64 bits are XORed with the $10^{th}$ and $11^{th}$ words.

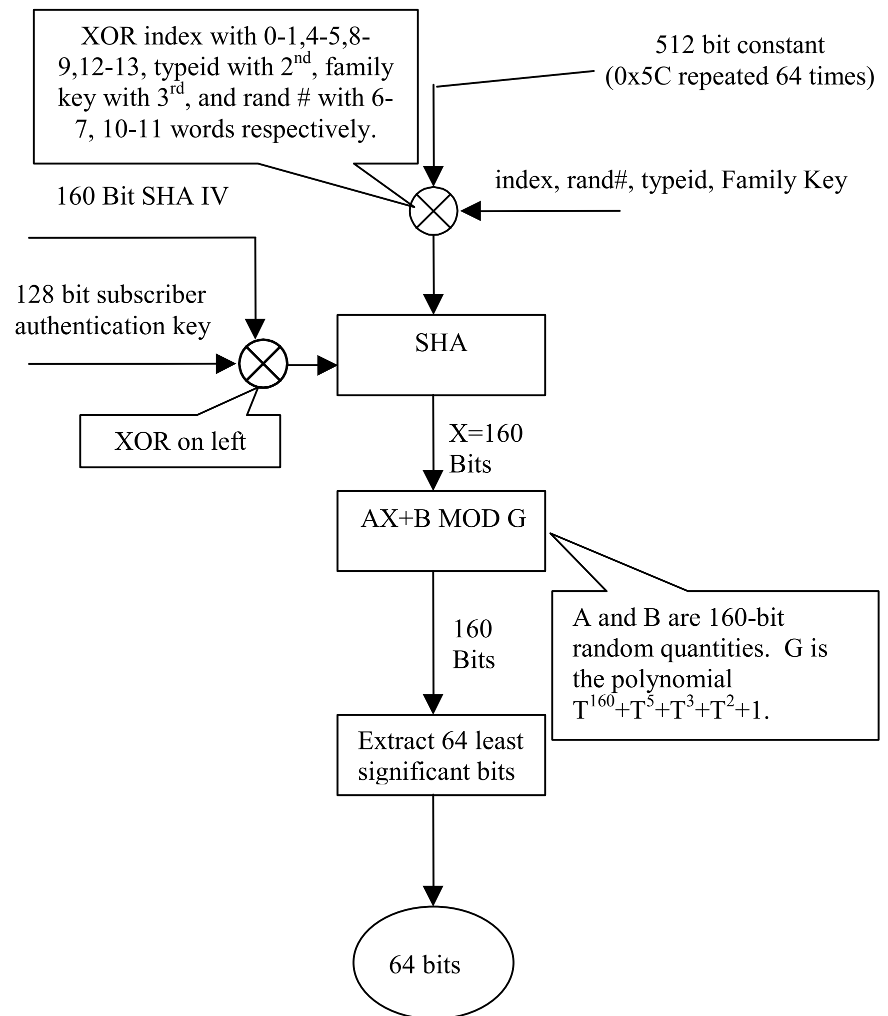
3. Run SHA to produce the 160-bit output.

4. The polynomial AX + B mod G is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T). A and B remain constant for the life of the UIM, and can be equal for all UIMs in the system.
   X is the 160-bit output from the SHA operation, treated as a polynomial with binary coefficients in the variable T.
   G is the polynomial $T^{160}+T^5+T^3+T^2+1$. Extract the least significant 64 bits and store it in the key buffer.

5. Steps 1 through 4 are repeated 2 times, with the index incremented between iterations. This gives a total of 128 bits. Store these 128 bits in CK.

This procedure is illustrated in Exhibit 2-2.

**Exhibit 2-2. Key Scheduler.**



#### *2.2.2.7.* Integrity Key (IK) Generation Procedures *f4*

Procedure name:

f4

Inputs from calling process:

| | |
|---|---|
| Subscriber Authentication Key (K) | 128 bits |
| Type Identifier (f4) | 8 bits |
| Random Number (RAND) | 128 bits |
| Family Key (Fmk) | 32 bits |

Inputs from internal stored data:

None.

Outputs to calling process:

Integrity Key (IK) 128 bits

Outputs to internal stored data:

None.

The function f4 is a pseudo random function used to generate the integrity key IK.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times
2. Load the subscriber authentication key and index value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. The 64-bit index value, initialized to 0, is XORed into the ($0^{th}$, $1^{st}$) words, ($4^{th}$, $5^{th}$) words, ($8^{th}$, $9^{th}$) words, and ($12^{th}$, $13^{th}$) words. $0^{th}$ is the least significant word and $15^{th}$ is the most significant. Next, a "type constant" (unique to function f4) is XORed into the $2^{nd}$ word, and the family key is XORed with the $3^{rd}$ word. The 128-bit random number is split into two parts (64 bits each). The least significant 64 bits are XORed with the $6^{th}$ and $7^{th}$ words, and the most significant 64 bits are XORed with the $10^{th}$ and $11^{th}$ words.
3. Run SHA to produce the 160-bit output.
4. The polynomial AX + B mod G is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T). A and B remain constant for the life of the UIM, and can be equal for all UIMs in the system.
   X is the 160-bit output from the SHA operation, treated as a polynomial with binary coefficients in the variable T.

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

5. Steps 1 through 4 are repeated 2 times, with the index incremented between iterations. This gives a total of 128 bits. Store these 128 bits in IK.

This procedure is illustrated in Exhibit 2-2.

#### 2.2.2.8. Anonymity Key (AK) Generation Procedure *f5*

Procedure name:

f5

Inputs from calling process:

| | |
|---|---|
| Subscriber Authentication Key (K) | 128 bits |
| Random Number (RAND) | 128 bits |
| Family Key (Fmk) | 32 bit |
| Type Identifier (f5) | 8 bits |

Inputs from internal stored data:

None.

Outputs to calling process:

| | |
|---|---|
| Anonymity Key (AK) | 48 bits |

Outputs to internal stored data:

None.

The function f5 is used to compute the anonymity key AK, which protects Authentication Vector sequence numbers.

Procedure:

1. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times

2. Load the subscriber authentication key and counter value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. A “type constant” (unique to function f5) is XORed into the 2nd word. The 32-bit family key is XORed into the 3rd word, 128-bit RAND is XORed into words 4-7th. Run SHA to produce a 160-bit output.

3. The polynomial (AX + B mod G) is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T). A and B remain constant for the life of the UIM, and can be equal for all

UIMs in the system.
X is the 160-bit output from the SHA operation, treated as a polynomial with binary coefficients in the variable T.
G is the polynomial $T^{160}+T^5+T^3+T^2+1$.

4. Only the 48 least significant bits are used.

#### 2.2.2.9. Resynchronization Anonymity Key (AKS) Generation Procedure *f5**

Procedure name:

f5*

Inputs from calling process:

| | |
|---|---|
| Subscriber Authentication Key (K) | 128 bits |
| Random Number (RAND) | 128 bits |
| Family Key (Fmk) | 32 bit |
| Type Identifier (f5*) | 8 bits |

Inputs from internal stored data:

None.

Outputs to calling process:

| | |
|---|---|
| Anonymity Key (AKS) | 48 bits |

Outputs to internal stored data:

None.

The function f5* is used to compute an anonymity key AK, which protects Authentication Vector sequence numbers in the resynchronization message.

Procedure:

5. Load the registers of SHA with known constants as follows:
   Load the IV with the standard SHA IV constant
   Load the 512-bit payload with the constant 0x5C repeated 64 times

6. Load the subscriber authentication key and counter value as follows:
   XOR the subscriber authentication key into the leftmost (most significant) 128 bits of IV. A “type constant” (unique to function f5) is XORed into the 2nd word. The 32-bit family key is XORed into the 3rd word, 128-bit RAND is XORed into words 4-7th. Run SHA to produce a 160-bit output.

7. The polynomial (AX + B mod G) is calculated, where:
   A and B are predetermined 160-bit random numbers (treated as polynomials with binary coefficients in the variable T). A and B remain constant for the life of the UIM, and can be equal for all

UIMs in the system.

X is the 160-bit output from the SHA operation, treated as a polynomial with binary coefficients in the variable T.

G is the polynomial $T^{160}+T^5+T^3+T^2+1$.

8. Only the 48 least significant bits are used.

# 2.3. Enhanced Voice and Data Privacy

## 2.3.1. TDMA (TIA-136)

Reserved.

## 2.3.2. CDMA (TIA/EIA/IS-2000)

### 2.3.2.1. 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 CMEAKEY (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.

When the Mobile Station performs authentication in accordance with air interface procedures that invoke AKA (see 2.2), the key used for the Rijndael encryption algorithm should be the 128-bit key CK generated using procedure **f3** (see 2.2.2.6).

### 2.3.2.2. Enhanced Privacy Algorithm

#### 2.3.2.2.1. Algorithm

The enhanced privacy algorithm is 128-bit Rijndael. Refer to 3.1 for a reference implementation of the enhanced privacy procedures using Rijndael.

#### 2.3.2.2.2. 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.

#### 2.3.2.2.3. ESP_maskbits Procedure

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.

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.

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.

### 2.3.2.2.4.ESP_AES Procedure

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.

# 3. Reference Implementations

## 3.1. CDMA Enhanced 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**

```
/* rijndael-alg-ref.h   v2.0   August '99
 * Reference ANSI C code
 * authors: Paulo Barreto
 *          Vincent Rijmen
 */
#ifndef __RIJNDAEL_ALG_H
#define __RIJNDAEL_ALG_H

#define MAXBC           (256/32)
#define MAXKC           (256/32)
#define MAXROUNDS          14

typedef unsigned char      word8;
typedef unsigned short     word16;
typedef unsigned long      word32;


int rijndaelKeySched (word8 k[4][MAXKC], int keyBits, int blockBits,
        word8 rk[MAXROUNDS+1][4][MAXBC]);
int rijndaelEncrypt (word8 a[4][MAXBC], int keyBits, int blockBits,
        word8 rk[MAXROUNDS+1][4][MAXBC]);
int rijndaelEncryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
        word8 rk[MAXROUNDS+1][4][MAXBC], int rounds);
int rijndaelDecrypt (word8 a[4][MAXBC], int keyBits, int blockBits,
        word8 rk[MAXROUNDS+1][4][MAXBC]);
int rijndaelDecryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
        word8 rk[MAXROUNDS+1][4][MAXBC], int rounds);

#endif /* __RIJNDAEL_ALG_H */
```

**Exhibit 3-2 Rijndael Box Data**

```
/* "boxes-ref.dat" */

word8 Logtable[256] = {
  0,   0,  25,   1,  50,   2,  26, 198,  75, 199,  27, 104,  51, 238, 223,   3,
100,   4, 224,  14,  52, 141, 129, 239,  76, 113,   8, 200, 248, 105,  28, 193,
125, 194,  29, 181, 249, 185,  39, 106,  77, 228, 166, 114, 154, 201,   9, 120,
101,  47, 138,   5,  33,  15, 225,  36,  18, 240, 130,  69,  53, 147, 218, 142,
150, 143, 219, 189,  54, 208, 206, 148,  19,  92, 210, 241,  64,  70, 131,  56,
102, 221, 253,  48, 191,   6, 139,  98, 179,  37, 226, 152,  34, 136, 145,  16,
126, 110,  72, 195, 163, 182,  30,  66,  58, 107,  40,  84, 250, 133,  61, 186,
 43, 121,  10,  21, 155, 159,  94, 202,  78, 212, 172, 229, 243, 115, 167,  87,
```

```
175,  88, 168,  80, 244, 234, 214, 116,  79, 174, 233, 213, 231, 230, 173, 232,
 44, 215, 117, 122, 235,  22,  11, 245,  89, 203,  95, 176, 156, 169,  81, 160,
127,  12, 246, 111,  23, 196,  73, 236, 216,  67,  31,  45, 164, 118, 123, 183,
204, 187,  62,  90, 251,  96, 177, 134,  59,  82, 161, 108, 170,  85,  41, 157,
151, 178, 135, 144,  97, 190, 220, 252, 188, 149, 207, 205,  55,  63,  91, 209,
 83,  57, 132,  60,  65, 162, 109,  71,  20,  42, 158,  93,  86, 242, 211, 171,
 68,  17, 146, 217,  35,  32,  46, 137, 180, 124, 184,  38, 119, 153, 227, 165,
103,  74, 237, 222, 197,  49, 254,  24,  13,  99, 140, 128, 192, 247, 112,   7,
};

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

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

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

```
word8 iG[4][4] = {
0x0e, 0x09, 0x0d, 0x0b,
0x0b, 0x0e, 0x09, 0x0d,
0x0d, 0x0b, 0x0e, 0x09,
0x09, 0x0d, 0x0b, 0x0e,
};

word32 rcon[30] = {
  0x01,0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
0xfa, 0xef, 0xc5, 0x91, };
```

**Exhibit 3-3 Rijndael Algorithm**

```
/* rijndael-alg-ref.c   v2.0   August '99
 * Reference ANSI C code
 * authors: Paulo Barreto
 *          Vincent Rijmen
 *
 * This code is placed in the public domain.
 */

#include <stdio.h>
#include <stdlib.h>

#include "rijndael-alg-ref.h"

#define SC   ((BC - 4) >> 1)

#include "boxes-ref.dat"

static word8 shifts[3][4][2] = {
   0, 0,
   1, 3,
   2, 2,
   3, 1,

   0, 0,
   1, 5,
   2, 4,
   3, 3,

   0, 0,
   1, 7,
   3, 5,
   4, 4
};


word8 mul(word8 a, word8 b) {
   /* multiply two elements of GF(2^m)
    * needed for MixColumn and InvMixColumn
    */
    if (a && b) return Alogtable[(Logtable[a] + Logtable[b])%255];
    else return 0;
}

void KeyAddition(word8 a[4][MAXBC], word8 rk[4][MAXBC], word8 BC) {
    /* Exor corresponding text input and round key input bytes
     */
    int i, j;

    for(i = 0; i < 4; i++)
       for(j = 0; j < BC; j++) a[i][j] ^= rk[i][j];
}
```

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

```
    case 192: KC = 6; break;
    case 256: KC = 8; break;
    default : return (-1);
    }

    switch (blockBits) {
    case 128: BC = 4; break;
    case 192: BC = 6; break;
    case 256: BC = 8; break;
    default : return (-2);
    }

    switch (keyBits >= blockBits ? keyBits : blockBits) {
    case 128: ROUNDS = 10; break;
    case 192: ROUNDS = 12; break;
    case 256: ROUNDS = 14; break;
    default : return (-3); /* this cannot happen */
    }


    for(j = 0; j < KC; j++)
        for(i = 0; i < 4; i++)
            tk[i][j] = k[i][j];
    t = 0;
    /* copy values into round key array */
    for(j = 0; (j < KC) && (t < (ROUNDS+1)*BC); j++, t++)
        for(i = 0; i < 4; i++) W[t / BC][i][t % BC] = tk[i][j];

    while (t < (ROUNDS+1)*BC) { /* while not enough round key material
calculated */
        /* calculate new values */
        for(i = 0; i < 4; i++)
            tk[i][0] ^= S[tk[(i+1)%4][KC-1]];
        tk[0][0] ^= rcon[rconpointer++];

        if (KC != 8)
            for(j = 1; j < KC; j++)
                for(i = 0; i < 4; i++) tk[i][j] ^= tk[i][j-1];
        else {
            for(j = 1; j < KC/2; j++)
                for(i = 0; i < 4; i++) tk[i][j] ^= tk[i][j-1];
            for(i = 0; i < 4; i++) tk[i][KC/2] ^= S[tk[i][KC/2 - 1]];
            for(j = KC/2 + 1; j < KC; j++)
                for(i = 0; i < 4; i++) tk[i][j] ^= tk[i][j-1];
    }
    /* copy values into round key array */
    for(j = 0; (j < KC) && (t < (ROUNDS+1)*BC); j++, t++)
        for(i = 0; i < 4; i++) W[t / BC][i][t % BC] = tk[i][j];
    }

    return 0;
}

int rijndaelEncrypt (word8 a[4][MAXBC], int keyBits, int blockBits, word8
rk[MAXROUNDS+1][4][MAXBC])
{
    /* Encryption of one block.
     */
    int r, BC, ROUNDS;

    switch (blockBits) {
    case 128: BC = 4; break;
    case 192: BC = 6; break;
    case 256: BC = 8; break;
    default : return (-2);
    }

```

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

```
    }

    /* if necessary, do the last, special, round:
     */
    if (rounds == ROUNDS) {
       Substitution(a,S,BC);
       ShiftRow(a,0,BC);
       KeyAddition(a,rk[ROUNDS],BC);
    }

    return 0;
}


int rijndaelDecrypt (word8 a[4][MAXBC], int keyBits, int blockBits, word8
rk[MAXROUNDS+1][4][MAXBC])
{
    int r, BC, ROUNDS;

    switch (blockBits) {
    case 128: BC = 4; break;
    case 192: BC = 6; break;
    case 256: BC = 8; break;
    default : return (-2);
    }

    switch (keyBits >= blockBits ? keyBits : blockBits) {
    case 128: ROUNDS = 10; break;
    case 192: ROUNDS = 12; break;
    case 256: ROUNDS = 14; break;
    default : return (-3); /* this cannot happen */
    }

    /* To decrypt: apply the inverse operations of the encrypt routine,
     *             in opposite order
     *
     * (KeyAddition is an involution: it 's equal to its inverse)
     * (the inverse of Substitution with table S is Substitution with the
inverse table of S)
     * (the inverse of Shiftrow is Shiftrow over a suitable distance)
     */

         /* First the special round:
     *   without InvMixColumn
     *   with extra KeyAddition
     */
    KeyAddition(a,rk[ROUNDS],BC);
    Substitution(a,Si,BC);
    ShiftRow(a,1,BC);

    /* ROUNDS-1 ordinary rounds
     */
    for(r = ROUNDS-1; r > 0; r--) {
       KeyAddition(a,rk[r],BC);
       InvMixColumn(a,BC);
       Substitution(a,Si,BC);
       ShiftRow(a,1,BC);
    }

    /* End with the extra key addition
     */

    KeyAddition(a,rk[0],BC);

    return 0;
}
```

```
int rijndaelDecryptRound (word8 a[4][MAXBC], int keyBits, int blockBits,
   word8 rk[MAXROUNDS+1][4][MAXBC], int rounds)
/* Decrypt only a certain number of rounds.
 * Only used in the Intermediate Value Known Answer Test.
 * Operations rearranged such that the intermediate values
 * of decryption correspond with the intermediate values
 * of encryption.
 */
{
   int r, BC, ROUNDS;

   switch (blockBits) {
   case 128: BC = 4; break;
   case 192: BC = 6; break;
   case 256: BC = 8; break;
   default : return (-2);
   }

   switch (keyBits >= blockBits ? keyBits : blockBits) {
   case 128: ROUNDS = 10; break;
   case 192: ROUNDS = 12; break;
   case 256: ROUNDS = 14; break;
   default : return (-3); /* this cannot happen */
   }


   /* make number of rounds sane */
   if (rounds > ROUNDS) rounds = ROUNDS;

        /* First the special round:
    *   without InvMixColumn
    *   with extra KeyAddition
    */
   KeyAddition(a,rk[ROUNDS],BC);
   Substitution(a,Si,BC);
   ShiftRow(a,1,BC);

   /* ROUNDS-1 ordinary rounds
    */
   for(r = ROUNDS-1; r > rounds; r--) {
      KeyAddition(a,rk[r],BC);
      InvMixColumn(a,BC);
      Substitution(a,Si,BC);
      ShiftRow(a,1,BC);
   }

   if (rounds == 0) {
      /* End with the extra key addition
       */
      KeyAddition(a,rk[0],BC);
   }

   return 0;
}
```

### 3.1.2. ESP Procedures

**Exhibit 3-4 Header for ESP**

```
/* "esp.h" Header for ESP. */
#ifndef ESP_HEADER
#define ESP_HEADER

#define KEYLENGTH 16 /* octets */
```

```
/* external interface declarations */
void ESP_privacykey(unsigned char key[KEYLENGTH]);
void
ESP_maskbits(unsigned char *fresh,
           int freshsize,
           unsigned char *buf,
           unsigned long bit_offset,
           unsigned long bit_count);
void
ESP_AES(   unsigned char key[KEYLENGTH],
           unsigned char *fresh,
           int freshsize,
           unsigned char *buf,
           unsigned long bit_offset,
           unsigned long bit_count);

#endif
```

**Exhibit 3-5 ESP_keysched and ESP_maskbits**

```
/* "esp.c" */

#include "esp.h"

#include "rijndael-alg-ref.h"
#define BLOCKLENGTH 16 /* octets */

/* internal storage */

/* ESP_privacykeyschedule consists of the array rk[][]. */

static
unsigned char   rk[MAXROUNDS+1][4][MAXBC]; /* rijndael Key Schedule */

/* Schedule key in internal storage. */

void
ESP_privacykey(unsigned char key[KEYLENGTH])
{
   unsigned char k[4][MAXKC];
   int           i;

   /* reshape key for rijndael */
   for (i = 0; i < KEYLENGTH; ++i)
      k[i%4][i/4] = key[i];
   rijndaelKeySched(k, KEYLENGTH*8, BLOCKLENGTH*8, rk);
}


/* encrypt/decrypt a buffer of data or output an encryption mask */

void
ESP_maskbits(unsigned char *fresh,
           int freshsize,
           unsigned char *buf,
           unsigned long bit_offset,
           unsigned long bit_count)
{
   int              i;
   unsigned long    counter;
   unsigned char    *bptr;
   unsigned char    b[4][MAXBC], offset, mask, bit_mask, mask_size, last_mask;

   if (bit_count <= 0)
      return;
```

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

```
            }
        }

        /* use the last bits in the Rijndael buffer, if any */
        if (offset != 0)
        {
            *bptr ^= (last_mask << (8 - offset)) & bit_mask;
            if (mask_size > offset)
                mask_size = offset;
            bit_count -= mask_size;
            if (bit_count <= 0)
                return;
        }
    }
}

void
ESP_AES(   unsigned char key[KEYLENGTH],
           unsigned char *fresh,
           int freshsize,
           unsigned char *buf,
           unsigned long bit_offset,
           unsigned long bit_count)
{
    ESP_privacykey(key);
    ESP_maskbits(fresh,freshsize,buf,bit_offset,bit_count);
}
```

## 3.2. SHA-Based AKA Functions

### 3.2.1. SHA-1

**Exhibit 3-6 SHA-1 Header**

```
/* "sha.h" */

#ifndef SHA_H
#define SHA_H

/* header for SHA and related procedures */
#define WORD unsigned long
#define DIGEST_LENGTH 20
typedef struct {
    unsigned char digest[DIGEST_LENGTH]; /* Message digest */
    WORD          count[2];             /* count of bits */
    WORD          data[16];             /* data buffer */
}
    SHA_INFO;

void shaInitial(SHA_INFO *sha_info);
void shaUpdate(SHA_INFO *sha_info,
               unsigned char *buffer,
               unsigned long offset,
               unsigned long count);
void shaFinal(SHA_INFO *sha_info);

#endif
```

**Exhibit 3-7 SHA-1 Code**

```
/* "sha.c" */

```

```
#include "sha.h"

static unsigned long A, B, C, D, E;

#define K1   0x5a827999
#define K2   0x6ed9eba1
#define K3   0x8f1bbcdc
#define K4   0xca62c1d6

#define S(a,n) ((a << n) | (a >> (32-n)))

static
unsigned char SHA_IV[20] = { 0x67, 0x45, 0x23, 0x01, 0xef, 0xcd, 0xab, 0x89,
                             0x98, 0xba, 0xdc, 0xfe, 0x10, 0x32, 0x54, 0x76,
                             0xc3, 0xd2, 0xe1, 0xf0 };

/* SHA ft(B,C,D) + Kt */

static unsigned long ftk(int t)
{
    if (t < 20)
        return( ((B & C) | (~B & D)) + K1 );
    else if (t < 40)
        return( (B ^ C ^ D) + K2 );
    else if (t < 60)
        return( ((B & C) | (B & D) | (C & D)) + K3 );
    else
        return( (B ^ C ^ D) + K4 );
}

/* the 80 rounds of SHA */

static
void shaHash(SHA_INFO *sha_info)
{
    unsigned long t, A0, B0, C0, D0, E0, W[16];
    int i, s;

    /* set the temporary digest values from the current digest,
       using shifts to ensure machine-independence */

    A  = (unsigned long)sha_info->digest[0] << 24;
    A += (unsigned long)sha_info->digest[1] << 16;
    A += (unsigned long)sha_info->digest[2] << 8;
    A += (unsigned long)sha_info->digest[3];
    B  = (unsigned long)sha_info->digest[4] << 24;
    B += (unsigned long)sha_info->digest[5] << 16;
    B += (unsigned long)sha_info->digest[6] << 8;
    B += (unsigned long)sha_info->digest[7];
    C  = (unsigned long)sha_info->digest[8] << 24;
    C += (unsigned long)sha_info->digest[9] << 16;
    C += (unsigned long)sha_info->digest[10] << 8;
    C += (unsigned long)sha_info->digest[11];
    D  = (unsigned long)sha_info->digest[12] << 24;
    D += (unsigned long)sha_info->digest[13] << 16;
    D += (unsigned long)sha_info->digest[14] << 8;
    D += (unsigned long)sha_info->digest[15];
    E  = (unsigned long)sha_info->digest[16] << 24;
    E += (unsigned long)sha_info->digest[17] << 16;
    E += (unsigned long)sha_info->digest[18] << 8;
    E += (unsigned long)sha_info->digest[19];

    /* save A-E */

    A0 = A;
    B0 = B;
    C0 = C;
```

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

```
/* initialize sha_info */

void shaInitial(SHA_INFO *sha_info)
{
   int i;

   /* set digest to its initial value.  Done one char at a time
      to ensure machine independence. */

   for (i = 0; i < 20; i++)
      sha_info->digest[i] = SHA_IV[i];

   /* clear data so that updates can be added in */

   for (i = 0; i < 16; i++)
      sha_info->data[i] = 0;

   /* set bit count to zero */

   sha_info->count[0] = sha_info->count[1] = 0;
}

/* update the digest using additional message data */

void shaUpdate(SHA_INFO *sha_info,
               unsigned char *buffer,
               unsigned long offset,
               unsigned long count)
{
   unsigned long data_count, t, mask_size;
   unsigned char *bptr, c, last, mask;

   /* enter the message data into the data buffer.  When the buffer
      is full (512 bits entered, update the digest and clear the
      data buffer for the next update.  */

   data_count = sha_info->count[1]%512;
   bptr = buffer + (offset/8);

   /* first fill the current octet of the buffer, so that
      the bit offset into the buffer is a multiple of 8 */
   last = *bptr++;
   if (data_count%8)
   {
      /* get a full byte from the buffer */
      c = last;
      last = *bptr++;
      if (offset%8)
      {
         c <<= (offset%8);
         c += last >> (8 - (offset%8));
      }

      /* set mask to fill the remaining bits of the octet */
      mask_size = 8 - (data_count%8);
      mask = 0xff << (data_count%8);

      /* adjust for short count */
      if (count < mask_size)
      {
         mask <<= (mask_size-count);
         mask_size = count;
      }

      /* store the bits */
      c = (c & mask) >> (data_count%8);
```

```
        sha_info->data[data_count/32] += (unsigned long)c << 8*(3 -
((data_count%32)/8));

        /* update count */
        t = sha_info->count[1];
        sha_info->count[1] += mask_size;
        if (sha_info->count[1] < t)
            sha_info->count[0]++;

        /* if the data buffer is full, update the digest */
        data_count += mask_size;
        if (data_count == 512)
        {
            shaHash(sha_info);
            data_count = 0;
        }

        /* start over with updated offset and count */
        offset += mask_size;
        count -= mask_size;
        bptr = buffer + (offset/8);
        last = *bptr++;
    }
    while (count != 0)
    {
        /* get the next full octet from the buffer */
        c = last;
        last = *bptr++;
        if (offset%8)
        {
            c <<= (offset%8);
            c += last >> (8 - (offset%8));
        }

        /* set mask to a full octet */
        mask_size = 8;
        mask = 0xff;

        /* adjust for short count */
        if (count < mask_size)
        {
            mask <<= (mask_size-count);
            mask_size = count;
        }

        /* store the bits */
        c &= mask;
        sha_info->data[data_count/32] += (unsigned long)c << 8*(3 -
((data_count%32)/8));

        /* update count */
        t = sha_info->count[1];
        sha_info->count[1] += mask_size;
        if (sha_info->count[1] < t)
            sha_info->count[0]++;

        /* if the data buffer is full, update the digest */
        data_count += mask_size;
        if (data_count == 512)
        {
            shaHash(sha_info);
            data_count = 0;
        }

        count -= mask_size;
    }
}
```

```
/* add pad bit of '1', zero fill and bit length, then update the digest */

void shaFinal(SHA_INFO *sha_info)
{
   /* add the pad bit of '1' */

   sha_info->data[(sha_info->count[1]%512)/32] +=
      1L << (31 - (sha_info->count[1]%32));

   /* if the data buffer is full, update the digest */

   if ((sha_info->count[1]%512) == 511)
      shaHash(sha_info);

   /* if there isn't room for 64 bits of bit length, leave the
      buffer zero filled to the end, update the digest and clear
      the buffer.  */

   if ((sha_info->count[1]%512) > (512-65))
      shaHash(sha_info);

   /* put in the bit length */

   sha_info->data[14] = sha_info->count[0];
   sha_info->data[15] = sha_info->count[1];

   /* update the digest */

   shaHash(sha_info);
}
```

### 3.2.2. AKA Functions f0-f5

**Exhibit 3-8 AKA Function Header**

```
/* aka.h */
/* header for SHA based AKA functions */

#ifndef AKA_H
#define AKA_H

#include "sha.h"

typedef unsigned char      uchar;
typedef unsigned short     word16;
typedef unsigned long int  word32;

#define L_KEY    16    /*128      bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_RAND   16    /*128      bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_FMK     4    /*32       bits */
#define L_SQN     6    /*48       bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_AMF     2    /*16       bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_MACA    8    /*64       bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_MACS    8    /*64       bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_RES    16    /*32<->128 bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_CK     16    /*128      bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_IK     16    /*128      bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_AK      6    /*48       bits 3G TS 33.105.v.3.0(2000-03)*/
#define L_AKS     6    /*48       bits 3G TS 33.105.v.3.0(2000-03)*/

/* function definitions */

void f0(uchar seed[],uchar fi,uchar Fmk[],uchar buff[]);
```

```
void f1(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar SQN[],uchar AMF[],uchar
MACA[]);
void f1star(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar SQN[],uchar
AMF[],uchar MACS[]);
void f2(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar RES[],int l_res);
void f3(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar *CK);
void f4(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar *IK);
void f5(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar AK[]);
void f5star(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar AKS[]);

#endif
```

**Exhibit 3-9 AKA Function Code**

```
/* aka.c:  sha based function for aka */

#include <string.h>
#include "aka.h"

static uchar counter[8]={0};

static uchar G[20] = {  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
                        0x00, 0x00, 0x00, 0x2d};
static uchar A[20] = {  0x9d, 0xe9, 0xc9, 0xc8, 0xef, 0xd5, 0x78, 0x11,
                        0x48, 0x23, 0x14, 0x01, 0x90, 0x1f, 0x2d, 0x49,
                        0x3f, 0x4c, 0x63, 0x65};
static uchar B[20] = {  0x75, 0xef, 0xd1, 0x5c, 0x4b, 0x8f, 0x8f, 0x51,
                        0x4e, 0xf3, 0xbc, 0xc3, 0x79, 0x4a, 0x76, 0x5e,
                        0x7e, 0xec, 0x45, 0xe0};
static
void modred(uchar *z,int shift,uchar *base);

/* This function performs the operation of (A*X+B) mod 2^160+2^5+2^3+2^2+1
 *
 *
*/

void whiten(uchar xx[])
{
   uchar z[40];
   int i, j;

   /* calculate A * X in polynomial form */
   for (i=0;i<40;i++)
      z[i]=0;

   for (i=0;i<20;i++)
   {
      for (j=0;j<8;j++)
      {
         if ((xx[i]<<j) & 0x80)
            modred(z,159-(i*8+j),A); /* z^=A<<(159-(i*8+j)) */
      }
   }


   /* AX MOD G done as modular reduction for bit 160 to 319 */
   for (i=0;i<20;i++)
   {
      for (j=0;j<8;j++)
      {
         if ((z[i]<<j)&0x80)
            modred(z,159-(i*8+j),G);
      }
   }
```

```
1
2       /* add B and copy back result */
3       for (i = 0; i < 20; i++)
4           xx[i] = z[i+20] ^ B[i];
5   }
6
7   /* This function perform the operation of shifting 320 bits and XOR.
8    *
9    *
10  */
11
12  static
13  void modred(uchar *z,int shift,uchar *base)
14  {
15      int byteshift, bitshift,i;
16      uchar q[21],yn,yn1;
17
18      for (i=0;i<20;i++)
19          q[i] = base[i];
20      q[20] = 0;
21
22      /* we divide into byte shifting and bit shifting */
23      byteshift = shift / 8;
24      bitshift = shift % 8;
25
26      /* do bit shifting */
27      if (bitshift != 0)
28      {
29          yn = 0;
30          for (i = 0; i <= 20; i++)
31          {
32              yn1 = yn;
33              yn = q[i];
34              q[i] >>= 8-bitshift;
35              q[i] |= yn1 << bitshift;
36          }
37          /* shift one more byte, since bits have effectively been
38             shifted into the next byte upward */
39          byteshift++;
40      }
41
42      /* z ^= q and send back result in z */
43      for (i = 0; i < 20; i++)
44          z[i+20-byteshift] ^= q[i];
45      if (bitshift != 0)
46          z[40-byteshift] ^= q[20];
47  }
48
49  /* This function performs generation of 64-bit pseudo random number RAND.
50   *
51   *
52  */
53
54  void
55  f0(uchar seed[],uchar fi,uchar Fmk[],uchar buff[])
56  {
57      SHA_INFO sha_info;
58      uchar buf[64];
59      uchar t;
60      int i;
61
62      shaInitial(&sha_info);
63      for (i = 0; i < L_KEY; i++)
64          sha_info.digest[i] ^= seed[i];
65
66      for (i = 0; i < 64; i++)
67          buf[i] = 0x5c;
```

```
    for (i = 0; i < 8; i++)
    {
       buf[i] ^= counter[i];
       buf[i+16] ^= counter[i];
       buf[i+32] ^= counter[i];
       buf[i+48] ^= counter[i];
    }

    buf[11] ^= fi;

    for (i = 0; i < 4; i++)
       buf[i+12] ^= Fmk[i];

    shaUpdate(&sha_info,buf,0,512);

    /* perform (AX+B)mod G */
    whiten(sha_info.digest);

    /* get 8 bytes or 64 bits */
    for (i=0;i<8;i++)
       buff[i] = sha_info.digest[i];

    /* increment counter */
    for (i = 7; i >= 0; i--)
    {
       t = counter[i];
       counter[i]++;
       if (counter[i] > t)
          break;
    }
}

/* This function performs generation of authentication signature MACA.
 *
 *
*/

void
f1(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar SQN[],uchar AMF[],uchar
MACA[])
{
    SHA_INFO sha_info;
    uchar buf[64];
    int i;

    /* NOTE: the following initialization of the sha_info struct can be
performed
       once when K is provisioned, and the results copied into sha_info at the
       start of this function. */
    shaInitial(&sha_info);
    for (i = 0; i < L_KEY; i++)
       sha_info.digest[i] ^= K[i];

    for (i = 0; i < 64; i++)
       buf[i] = 0x5c;

    for (i = 0; i < 4; i++)
       buf[i+12] ^= Fmk[i];
    for (i = 0; i < 16; i++)
       buf[i+16] ^= RAND[i];
    for (i = 0; i < 6; i++)
       buf[i+34] ^= SQN[i];
    for (i = 0; i < 2; i++)
       buf[i+42] ^= AMF[i];

    buf[11] ^= fi;
```

```
    shaUpdate(&sha_info,buf,0,512);

    /* perform (AX+B)mod G */
    whiten(sha_info.digest);

    for (i=0;i<L_MACA;i++)
        MACA[i] = sha_info.digest[i];
}

/* This function performs generation of resync authentication signature MACS.
 *
 *
*/

void
f1star(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar SQN[],uchar AMF[],uchar
MACS[])
{
    SHA_INFO sha_info;
    uchar buf[64];
    int i;

    /* NOTE: the following initialization of the sha_info struct can be
performed
        once when K is provisioned, and the results copied into sha_info at the
        start of this function. */
    shaInitial(&sha_info);
    for (i = 0; i < L_KEY; i++)
        sha_info.digest[i] ^= K[i];

    for (i = 0; i < 64; i++)
        buf[i] = 0x5c;

    for (i = 0; i < 4; i++)
        buf[i+12] ^= Fmk[i];
    for (i = 0; i < 16; i++)
        buf[i+16] ^= RAND[i];
    for (i = 0; i < 6; i++)
        buf[i+34] ^= SQN[i];
    for (i = 0; i < 2; i++)
        buf[i+42] ^= AMF[i];

    buf[11] ^= fi;

    shaUpdate(&sha_info,buf,0,512);

    /* perform (AX+B)mod G */
    whiten(sha_info.digest);

    for (i=0;i<L_MACS;i++)
        MACS[i] = sha_info.digest[i];
}

/* This function performs generation of user response RES.
 *
 *
*/

void
f2(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar RES[],int l_res)
{
    SHA_INFO sha_info;
    uchar j,buf[64];
    int i;

    if (l_res < 1)
```

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

```
        for (i = 0; i < 4; i++)
            buf[i+12] ^= Fmk[i];
        for (i = 0; i < 16; i++)
            buf[i+24] ^= RAND[i];

        buf[3] ^= j;
        buf[11] ^= fi;
        buf[19] ^= j;
        buf[35] ^= j;
        buf[51] ^= j;

        shaUpdate(&sha_info,buf,0,512);

        whiten(sha_info.digest);
        for (i=0;i<8;i++)
            CK[8*j+i] = sha_info.digest[i];
    }
}

/* This function performs generation of integrity key IK.
 *
 *
*/

void
f4(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar *IK)
{
    SHA_INFO sha_info;
    uchar j,buf[64];
    int i;

    for (j = 0; j < 2; j++)
    {
        /* NOTE: the following initialization of the sha_info struct can be
performed
           once when K is provisioned, and the results copied into sha_info at
the
           start of this loop. */
        shaInitial(&sha_info);
        for (i = 0; i < L_KEY; i++)
            sha_info.digest[i] ^= K[i];

        for (i = 0; i < 64; i++)
            buf[i] = 0x5c;

        for (i = 0; i < 4; i++)
            buf[i+12] ^= Fmk[i];
        for (i = 0; i < 16; i++)
            buf[i+24] ^= RAND[i];

        buf[3] ^= j;
        buf[11] ^= fi;
        buf[19] ^= j;
        buf[35] ^= j;
        buf[51] ^= j;

        shaUpdate(&sha_info,buf,0,512);

        whiten(sha_info.digest);
        for (i=0;i<8;i++)
            IK[8*j+i] = sha_info.digest[i];
    }
}

/* This function performs generation of anonymity key AK.
 *
 *
```

```
*/

void
f5(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar AK[])
{
   SHA_INFO sha_info;
   uchar buf[64];
   int i;

   /* NOTE: the following initialization of the sha_info struct can be
performed
      once when K is provisioned, and the results copied into sha_info at the
      start of this function. */
   shaInitial(&sha_info);
   for (i = 0; i < L_KEY; i++)
      sha_info.digest[i] ^= K[i];

   for (i = 0; i < 64; i++)
      buf[i] = 0x5c;

   for (i = 0; i < 4; i++)
      buf[i+12] ^= Fmk[i];
   for (i = 0; i < 16; i++)
      buf[i+16] ^= RAND[i];

   buf[11] ^= fi;

   shaUpdate(&sha_info,buf,0,512);

   /* perform (AX+B)mod G */
   whiten(sha_info.digest);

   for (i=0;i<L_AK;i++)
      AK[i] = sha_info.digest[i];
}

/* This function performs generation of anonymity key AK for
 * resynchronization.
 *
 *
*/

void
f5star(uchar K[],uchar fi,uchar *RAND,uchar Fmk[],uchar AKS[])
{
   SHA_INFO sha_info;
   uchar buf[64];
   int i;

   /* NOTE: the following initialization of the sha_info struct can be
performed
      once when K is provisioned, and the results copied into sha_info at the
      start of this function. */
   shaInitial(&sha_info);
   for (i = 0; i < L_KEY; i++)
      sha_info.digest[i] ^= K[i];

   for (i = 0; i < 64; i++)
      buf[i] = 0x5c;

   for (i = 0; i < 4; i++)
      buf[i+12] ^= Fmk[i];
   for (i = 0; i < 16; i++)
      buf[i+16] ^= RAND[i];

   buf[11] ^= fi;

```

```
    shaUpdate(&sha_info,buf,0,512);

    /* perform (AX+B)mod G */
    whiten(sha_info.digest);

    for (i=0;i<L_AKS;i++)
        AKS[i] = sha_info.digest[i];
}
```

# 4. Test Vectors

## 4.1. CDMA Enhanced 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**

```
bit_offset = 0; bit_count = 328

ad 23 08 ad 19 1d 93 71 d9 50 f4 d7 a3 a1 48 0c
7b 9c ce 3d 62 9a 33 39 61 67 e6 a2 a0 ec 3c c6
7b 3a 2a 73 b5 f8 9b 0a 98

bit_offset = 9; bit_count = 318

00 56 91 84 56 8c 8e c9 b8 ec a8 7a 6b d1 d0 a4
06 3d ce 67 1e b1 4d 19 9c b0 b3 f3 51 50 76 1e
63 3d 9d 15 39 da fc 4d 84

bit_offset = 5; bit_count = 320

05 69 18 45 68 c8 ec 9b 8e ca 87 a6 bd 1d 0a 40
63 dc e6 71 eb 14 d1 99 cb 0b 3f 35 15 07 61 e6
33 d9 d1 53 9d af c4 d8 50

bit_offset = 3; bit_count = 259

15 a4 61 15 a3 23 b2 6e 3b 2a 1e 9a f4 74 29 01
8f 73 99 c7 ac 53 46 67 2c 2c fc d4 54 1d 87 98
cc 00 00 00 00 00 00 00 00
```

### 4.1.2. Test Program

**Exhibit 4-2 Rijndael Test Program**

```
#include <stdio.h>
#include "esp.h"

unsigned char *tkey = "Test key 128bits";
unsigned char buf[41];
unsigned char fresh[8] = { 0, 0, 0, 0, 0, 0, 0, 1 };

void pause(void)
{
    printf("Press Enter to continue\n");
    getchar();
}

void main(void)
{
    int i;
```

```
    for (i = 0; i < 41; i++)
        buf[i] = 0;

    /* bit_offset = 0; bit_count = 8*41 */

    ESP_privacykey(tkey);
    ESP_maskbits(fresh,8,buf,0,8*41);

    printf("bit_offset = %d; bit_count = %d\n\n",0,8*41);

    for (i = 0; i < 16; i++)
        printf("%02x ",buf[i]);
    printf("\n");
    for (i = 16; i < 32; i++)
        printf("%02x ",buf[i]);
    printf("\n");
    for (i = 32; i < 41; i++)
        printf("%02x ",buf[i]);
    printf("\n");

    pause();

    /* bit_offset = 9, bit_count = 7+8*39 */

    printf("bit_offset = %d; bit_count = %d\n\n",9,8*39+6);

    for (i = 0; i < 41; i++)
        buf[i] = 0;

    ESP_privacykey(tkey);
    ESP_maskbits(fresh,8,buf,9,8*39+6);

    for (i = 0; i < 16; i++)
        printf("%02x ",buf[i]);
    printf("\n");
    for (i = 16; i < 32; i++)
        printf("%02x ",buf[i]);
    printf("\n");
    for (i = 32; i < 41; i++)
        printf("%02x ",buf[i]);
    printf("\n");

    pause();

    /* bit_offset = 5; bit_count = 8*40 */

    printf("bit_offset = %d; bit_count = %d\n\n",5,8*40);

    for (i = 0; i < 41; i++)
        buf[i] = 0;

    ESP_privacykey(tkey);
    ESP_maskbits(fresh,8,buf,5,8*40);

    for (i = 0; i < 16; i++)
        printf("%02x ",buf[i]);
    printf("\n");
    for (i = 16; i < 32; i++)
        printf("%02x ",buf[i]);
    printf("\n");
    for (i = 32; i < 41; i++)
        printf("%02x ",buf[i]);
    printf("\n");

    pause();
```

```
    /* bit_offset = 3; bit_count = 8*32+3 */

    printf("bit_offset = %d; bit_count = %d\n\n",3,8*32+3);

    for (i = 0; i < 41; i++)
       buf[i] = 0;

    ESP_privacykey(tkey);
    ESP_maskbits(fresh,8,buf,3,8*32+3);

    for (i = 0; i < 16; i++)
       printf("%02x ",buf[i]);
    printf("\n");
    for (i = 16; i < 32; i++)
       printf("%02x ",buf[i]);
    printf("\n");
    for (i = 32; i < 41; i++)
       printf("%02x ",buf[i]);
    printf("\n");

    pause();
}
```

## 4.2. SHA-Based Functions for AKA

### 4.2.1. Test Program Output

**Exhibit 4-3 AKA Function Test Output**

```
test vector for f0:
input section
seed is:  b0 ab b9 9d 6a c6 a7 4e b9 8e b6 c2 da b1 a5 51
fi0 is:      41
Fmk is:      41 48 41 47

output section
f0 RAND:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e


test vector for f1:
input section
K is:     ad 1b 5a 15 9b e8 6b 2c a6 6c 7a e4 0b ba 9b 9d
fi1 is:      42
RAND is:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
Fmk is:      41 48 41 47
SQN is:      00 00 00 00 00 01
AMF is:      00 01

output section
f1 MACA:  6a bd c4 da 73 c6 1b 8d


test vector for f1*:
input section
K is:     ad 1b 5a 15 9b e8 6b 2c a6 6c 7a e4 0b ba 9b 9d
fi1star is:  43
RAND is:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
Fmk is:      41 48 41 47
SQN is:      00 00 00 00 00 01
AMF is:      00 01

output section
f1*   MACS:  b0 17 35 9d 5d a8 81 a0
```

```
test vector for f2:
input section
K is:     ad 1b 5a 15 9b e8 6b 2c a6 6c 7a e4 0b ba 9b 9d
fi2 is:      44
RAND is:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
Fmk is:      41 48 41 47

output section
f2 RES:   d8 2e 28 2a dc 13 c0 f1 68 65 66 33 9b f2 7e b6


test vector for f3:
input section
K is:     ad 1b 5a 15 9b e8 6b 2c a6 6c 7a e4 0b ba 9b 9d
fi3 is:      45
RAND is:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
Fmk is:      41 48 41 47

output section
f3 CK: 6e fd d8 32 f6 ff d4 dc a8 4a 54 96 fa 6e 29 93


test vector for f4:
input section
K is:     ad 1b 5a 15 9b e8 6b 2c a6 6c 7a e4 0b ba 9b 9d
fi4 is:      46
RAND is:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
Fmk is:      41 48 41 47

output section
f4 IK: c1 43 65 25 fa 60 7f 17 92 fc a8 9f b2 a7 bc 4a


test vector for f5
input section
K is:     ad 1b 5a 15 9b e8 6b 2c a6 6c 7a e4 0b ba 9b 9d
fi5 is:      47
RAND is:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
Fmk is:      41 48 41 47

output section
f5 AK: 59 4c c7 c1 7c 06


test vector for f5*
input section
K is:     ad 1b 5a 15 9b e8 6b 2c a6 6c 7a e4 0b ba 9b 9d
fi5* is:  48
RAND is:  4b 05 2b 20 e2 a0 6c 8f f7 00 da 51 2b 4e 11 1e
Fmk is:      41 48 41 47

output section
f5*   AKS:   b2 d6 37 36 5c ea
```

### 4.2.2. Test Program

**Exhibit 4-4 AKA Function Test Program**

```
/* test_sha_based_aka.c */

#include <stdio.h>
#include "aka.h"

unsigned char *msg_data =
   "abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq";

void pause(void)
{
   printf("Press any key to continue\n");
   getchar();
}

void main()
{
   uchar K[]={0xad,0x1b,0x5a,0x15,0x9b,0xe8,0x6b,0x2c,
         0xa6,0x6c,0x7a,0xe4,0x0b,0xba,0x9b,0x9d};

   uchar seed[]={0xb0,0xab,0xb9,0x9d,0x6a,0xc6,0xa7,0x4e,
         0xb9,0x8e,0xb6,0xc2,0xda,0xb1,0xa5,0x51};

   uchar Fmk[L_FMK] = { 'A', 'H', 'A', 'G' };
   uchar RAND[L_RAND];

   uchar CK[L_CK];
   uchar IK[L_IK];

   uchar MACA[L_MACA];
   uchar MACS[L_MACS];
   uchar RES[L_RES];
   uchar AK[L_AK];
   uchar AKS[L_AKS];

   uchar SQN[L_SQN]={0x00,0x00,0x00,0x00,0x00,0x01};
   uchar fi0,fi1,fi1star,fi2,fi3,fi4,fi5,fi5star;

   uchar buff1[L_RAND/2],buff2[L_RAND/2];

   uchar AMF[2];

   int i;

   fi0=0x41;
   fi1=0x42;
   fi1star=0x43;
   fi2=0x44;
   fi3=0x45;
   fi4=0x46;
   fi5=0x47;
   fi5star=0x48;

   printf("test vector for f0:\n");
   printf("input section\n");
   printf("seed is: ");
   for(i=0;i<L_KEY;i++)
      printf("%02x ",seed[i]);
   printf("\n");
   printf("fi0 is:     %02x\n",fi0);
   printf("Fmk is:     ");
   for (i=0;i<L_FMK;i++)
```

```
1           printf("%02x ",Fmk[i]);
2       printf("\n");
3
4       printf("\n");
5       f0(seed,fi0,Fmk,buff1);
6       f0(seed,fi0,Fmk,buff2);
7       printf("output section\n");
8       printf("f0   RAND:  ");
9       for (i=0;i<L_RAND/2;i++)
10          printf("%02x ",buff1[i]);
11      for (i=0;i<L_RAND/2;i++)
12          printf("%02x ",buff2[i]);
13      printf("\n");
14
15      pause();
16
17      /* reuse RAND generated for the subsequent function calls*/
18      for (i=0;i<L_RAND/2;i++)
19          RAND[i] = buff1[i];
20      for (i=0;i<L_RAND/2;i++)
21          RAND[i+L_RAND/2] = buff2[i];
22
23      AMF[0]=0x00;
24      AMF[1]=0x01;
25
26      printf("\n");
27      printf("\n");
28      printf("test vector for f1:\n");
29      printf("input section\n");
30      printf("K is:    ");
31      for(i=0;i<L_KEY;i++)
32          printf("%02x ",K[i]);
33      printf("\n");
34      printf("fi1 is:     %02x\n",fi1);
35      printf("RAND is: ");
36      for(i=0;i<L_RAND;i++)
37          printf("%02x ",RAND[i]);
38      printf("\n");
39      printf("Fmk is:     ");
40      for(i=0;i<L_FMK;i++)
41          printf("%02x ",Fmk[i]);
42      printf("\n");
43      printf("SQN is:     ");
44      for(i=0;i<L_SQN;i++)
45          printf("%02x ",SQN[i]);
46      printf("\n");
47      printf("AMF is:     %02x %02x\n",AMF[0],AMF[1]);
48
49      f1(K,fi1,RAND,Fmk,SQN,AMF,MACA);
50      printf("\n");
51      printf("output section\n");
52      printf("f1   MACA:  ");
53      for (i=0;i<L_MACA;i++)
54          printf("%02x ",MACA[i]);
55      printf("\n");
56
57      pause();
58
59      printf("\n");
60      printf("\n");
61      printf("test vector for f1*:\n");
62      printf("input section\n");
63      printf("K is:    ");
64      for(i=0;i<L_KEY;i++)
65          printf("%02x ",K[i]);
66      printf("\n");
67      printf("fi1star is: %02x\n",fi1star);
```

```
1        printf("RAND is: ");
2        for(i=0;i<L_RAND;i++)
3           printf("%02x ",RAND[i]);
4        printf("\n");
5        printf("Fmk is:     ");
6        for(i=0;i<L_FMK;i++)
7           printf("%02x ",Fmk[i]);
8        printf("\n");
9        printf("SQN is:     ");
10       for(i=0;i<L_SQN;i++)
11          printf("%02x ",SQN[i]);
12       printf("\n");
13       printf("AMF is:     %02x %02x\n",AMF[0],AMF[1]);
14
15       printf("\n");
16       f1star(K,fi1star,RAND,Fmk,SQN,AMF,MACS);
17       printf("output section\n");
18       printf("f1*  MACS:  ");
19       for(i=0;i<L_MACS;i++)
20          printf("%02x ",MACS[i]);
21       for(i=0;i<L_SQN;i++)
22          SQN[i]=0x00;
23       AMF[0]=AMF[1]=0;
24       printf("\n");
25
26       pause();
27
28       printf("\n");
29       printf("\n");
30       printf("test vector for f2:\n");
31       printf("input section\n");
32       printf("K is:    ");
33       for(i=0;i<L_KEY;i++)
34          printf("%02x ",K[i]);
35       printf("\n");
36       printf("fi2 is:     %02x\n",fi2);
37       printf("RAND is: ");
38       for(i=0;i<L_RAND;i++)
39          printf("%02x ",RAND[i]);
40       printf("\n");
41       printf("Fmk is:     ");
42       for(i=0;i<L_FMK;i++)
43          printf("%02x ",Fmk[i]);
44       printf("\n");
45       printf("\n");
46       f2(K,fi2,RAND,Fmk,RES,L_RES);
47       printf("output section\n");
48       printf("f2   RES:   ");
49       for (i=0;i<L_RES;i++)
50          printf("%02x ",RES[i]);
51       printf("\n");
52
53       pause();
54
55       printf("\n");
56       printf("\n");
57       printf("test vector for f3:\n");
58       printf("input section\n");
59       printf("K is:    ");
60       for(i=0;i<L_KEY;i++)
61          printf("%02x ",K[i]);
62       printf("\n");
63       printf("fi3 is:     %02x\n",fi3);
64       printf("RAND is: ");
65       for(i=0;i<L_RAND;i++)
66          printf("%02x ",RAND[i]);
67       printf("\n");
```

```
1        printf("Fmk is:      ");
2        for(i=0;i<L_FMK;i++)
3           printf("%02x ",Fmk[i]);
4        printf("\n");
5
6        printf("\n");
7        f3(K,fi3,RAND,Fmk,CK);
8        printf("output section\n");
9        printf("f3    CK: ");
10       for (i=0;i<L_CK;i++)
11          printf("%02x ",CK[i]);
12       printf("\n");
13
14       pause();
15
16       printf("\n");
17       printf("\n");
18       printf("test vector for f4:\n");
19       printf("input section\n");
20       printf("K is:    ");
21       for(i=0;i<L_KEY;i++)
22          printf("%02x ",K[i]);
23       printf("\n");
24       printf("fi4 is:      %02x\n",fi4);
25       printf("RAND is: ");
26       for(i=0;i<L_RAND;i++)
27          printf("%02x ",RAND[i]);
28       printf("\n");
29       printf("Fmk is:      ");
30       for(i=0;i<L_FMK;i++)
31          printf("%02x ",Fmk[i]);
32       printf("\n");
33
34       printf("\n");
35       f4(K,fi4,RAND,Fmk,IK);
36       printf("output section\n");
37       printf("f4    IK: ");
38       for (i=0;i<L_IK;i++)
39          printf("%02x ",IK[i]);
40       printf("\n");
41
42       pause();
43
44       printf("\n");
45       printf("\n");
46       printf("test vector for f5\n");
47       printf("input section\n");
48       printf("K is:    ");
49       for(i=0;i<L_KEY;i++)
50          printf("%02x ",K[i]);
51       printf("\n");
52       printf("fi5 is:      %02x\n",fi5);
53       printf("RAND is: ");
54       for(i=0;i<L_RAND;i++)
55          printf("%02x ",RAND[i]);
56       printf("\n");
57       printf("Fmk is:      ");
58       for(i=0;i<L_FMK;i++)
59          printf("%02x ",Fmk[i]);
60       printf("\n");
61       printf("\n");
62       f5(K,fi5,RAND,Fmk,AK);
63       printf("output section\n");
64       printf("f5    AK: ");
65       for (i=0;i<L_AK;i++)
66          printf("%02x ",AK[i]);
67       printf("\n");
```

```
1
2       pause();
3
4       printf("\n");
5       printf("\n");
6       printf("test vector for f5*\n");
7       printf("input section\n");
8       printf("K is:    ");
9       for(i=0;i<L_KEY;i++)
10         printf("%02x ",K[i]);
11      printf("\n");
12      printf("fi5* is: %02x\n",fi5star);
13      printf("RAND is: ");
14      for(i=0;i<L_RAND;i++)
15         printf("%02x ",RAND[i]);
16      printf("\n");
17      printf("Fmk is:     ");
18      for(i=0;i<L_FMK;i++)
19         printf("%02x ",Fmk[i]);
20      printf("\n");
21      printf("\n");
22      f5star(K,fi5star,RAND,Fmk,AKS);
23      printf("output section\n");
24      printf("f5*  AKS:   ");
25      for (i=0;i<L_AKS;i++)
26         printf("%02x ",AKS[i]);
27      printf("\n");
28
29      pause();
30   }
31
32
```