TR45.AHAG

Common Cryptographic Algorithms, Revision D.1

Publication Version

September 13, 2000

NOTICE

EIA/TIA Engineering Standards and Publications are designed to serve the public interest through eliminating misunderstandings between manufacturers and purchasers, facilitating inter-changeability and improvement of products, and assisting the purchaser in selecting and obtaining with minimum delay the proper product for their particular need. Existence of such Standards and Publications shall not in any respect preclude any member or non-member of EIA or TIA from manufacturing or selling products not conforming to such Standards and Publications, nor shall the existence of such Standards and Publications preclude their voluntary use by those other than EIA or TIA members, whether the standard is to be used either domestically or internationally.

Standards and Publications are adopted by EIA/TIA without regard to whether or not their adoption may involve patents or articles, materials, or processes. By such action, EIA/TIA does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the Recommended Standard or Publication.

TIA TR45 Ad Hoc Authentication Group Documents

TIA TR45 Ad Hoc Authentication Group Documents contain information deemed to be of technical value to the industry, and are published at the request of the TR45 Ad Hoc Authentication Group without necessarily following the rigorous public review and resolution of comments which is a procedural part of the development of an EIA/TIA Standard.

TIA TR45 Ad Hoc Authentication Group Documents bear on or are subject to the export jurisdiction of the US Department of Commerce as specified in Export Administration Regulations (EAR), Title 15 CFR parts 730 through 774 inclusive. An export license may be required for the transmission of such material in any form outside of the United States of America.

Contact

TELECOMMUNICATIONS INDUSTRY ASSOCIATION
Engineering Department
2500 Wilson Boulevard, Suite 300
Arlington, VA 22201
Copyright 2000
TELECOMMUNICATIONS INDUSTRY ASSOCIATION
All rights reserved
Printed in the United States

Document History

Revision	Date	Remarks
0	02-05-93	Frozen for PN-3118 Ballot
0.1	04-21-93	Adopted by TR45 AHAG
A	12-14-94	Major revision, incorporating ORYX data encryption algorithms and ANSI C algorithm descriptions
A.1	04-25-95	Corrections to ORYX algorithm and test vectors; conversion to Word 6.0
В	04-26-95	Add procedures for wireless residential extension authentication
B.1	04-15-97	Version for PN-3795 ballot.
С	10-27-98	Add ECMEA and related key management procedures
D	03-14-00	Add SCEMA and related procedures
D.1	09-13-00	Corrections to SCEMA key scheduling

Table of Contents

2	1. INT	RODUCTION	1
3	1.1.	Notations	2
4	1.2.	Definitions	2
5	2. PR	DCEDURES	6
6	2.1.	CAVE	6
7	2.2.	Authentication Key (A-Key) Procedures	17
8	2.2.1		17
9	2.2.2	A-Key Verification	21
10	2.3.	SSD Generation and Update	24
11	2.3.1		24
12	2.3.2	SSD Update Procedure	27
13	2.4.	Authentication Signature Calculation Procedure	28
14	2.5.	Secret Key and Secret Parameter Generation	33
15	2.5.1	. CMEA Encryption Key and VPM Generation Procedure	34
16		5.1.1. CMEA key Generation	35
17		5.1.2. Voice Privacy Mask Generation	36
18	2.5.2	$\boldsymbol{\varepsilon}$	40
19	2.5.3		45
20	2.5.4	ECMEA Secrets Generation for Non-Financial Messages Procedure	48
21		Message Encryption/Decryption Procedures	52
22	2.6.1	71 71	52
23	2.6.2	ECMEA Encryption/Decryption Procedure	55
24	2.7.	Wireless Residential Extension Procedures	65
25	2.7.1	. WIKEY Generation	66
26	2.7.2		69
27	2.7.3	. Wireline Interface Authentication Signature Calculation Procedure	72
28	2.7.4	Wireless Residential Extension Authentication Signature Calculation Procedure	76
29	2.8.	Basic Wireless Data Encryption	79
30	2.8.1		82
31	2.8.2	. L-Table Generation Procedure	86
32	2.8.3	Data Encryption Mask Generation Procedure	88
33	2.9.	Enhanced Voice and Data Privacy	90
34	2.9.1		90
35		2.1.1. DTC Key Generation	91
36		2.1.2. DCCH Key Generation	93
37		2.1.3. SCEMA Secret Generation	95
38	2.9.2		100
39	2.9.3	. SCEMA Encryption/Decryption Procedure (Level 1)	103

1	2.9.4. Block and KSG Encryption Primitives (Level 2)	111
2	2.9.4.1. SCEMA KSG	111
3	2.9.4.2. Long Block Encryptor	113
4	2.9.4.3. Short Block Encryptor	116
5	2.9.5. Voice, Message, and Data Encryption Procedures (Level 3)	122
6	2.9.5.1. Enhanced Voice Privacy	122
7	2.9.5.2. Enhanced Message Encryption	128
8	2.9.5.3. Enhanced Wireless Data Encryption	133
9	3. TEST VECTORS	136
9	0. 1201 V2010N0	100
10	3.1. CAVE Test Vectors	136
11	3.1.1. Vector 1	136
12	3.1.2. Vector 2	137
13	3.1.3. Vector 3	138
14	3.1.4. Test Program	139
15	3.2. Wireless Residential Extension Test Vector	146
16	3.2.1. Input data	146
17	3.2.2. Test Program	147
18	3.2.3. Test Program Output	148
19	3.3. Basic Data Encryption Test Vector	149
20	3.3.1. Input data	149
21	3.3.2. Test Program	149
22	3.3.3. Test Program Output	151
23	3.4. Enhanced Voice and Data Privacy Test Vectors	152
24	3.4.1. Input Data	152
25	3.4.2. Test Program	152
26	3.4.2.1. Main program file	152
27	3.4.2.2. Vector set 3	155
28	3.4.2.3. Vector set 4	157
29	3.4.2.4. Vector set 5	158
30	3.4.2.5. Vector set 6	162
31	3.4.2.6. Vector set 7	166
32	3.4.2.7. Vector set 8	170
33	3.4.3. Test Program Input and Output	170

Common Cryptographic Algorithms Revision D.1

09/13/2000

List of Exhibits

2	EXHIBIT 2-1 CAVE ELEMENTS	
3	EXHIBIT 2-2 CAVE ALGORITHM EXTERNAL HEADER	8
4	EXHIBIT 2-3 CAVE ALGORITHM INTERNAL HEADER	10
5	EXHIBIT 2-4 CAVE ALGORITHM	11
6	EXHIBIT 2-5 CAVE TABLE	16
7	EXHIBIT 2-6 CAVE INITIAL LOADING FOR A-KEY CHECKSUM	18
8	EXHIBIT 2-7 A-KEY CHECKSUM	19
9	EXHIBIT 2-8 A-KEY VERIFICATION	
10	EXHIBIT 2-9 GENERATION OF SSD A NEW AND SSD B NEW	25
11	EXHIBIT 2-10 SSD GENERATION	
12	EXHIBIT 2-11 SSD UPDATE	27
13	EXHIBIT 2-12 CAVE INITIAL LOADING FOR AUTHENTICATION SIGNATURES	29
14	EXHIBIT 2-13 CALCULATION OF AUTH_SIGNATURE	30
15	EXHIBIT 2-14 CODE FOR CALCULATION OF AUTH SIGNATURE	31
16	EXHIBIT 2-15 CMEA KEY AND VPM GENERATION	
17	EXHIBIT 2-16 GENERATION OF CMEA KEY AND VPM	
18	EXHIBIT 2-17 DETAILED GENERATION OF CMEA KEY AND VPM	
19	EXHIBIT 2-18 GENERATION OF ECMEA SECRETS	
20	EXHIBIT 2-19 ECMEA SECRET GENERATION	
21	EXHIBIT 2-20 GENERATION OF NON-FINANCIAL SEED KEY	
22	EXHIBIT 2-21 NON-FINANCIAL SEED KEY GENERATION	
23	EXHIBIT 2-22 GENERATION OF NON-FINANCIAL SECRETS	
24	EXHIBIT 2-23 NON-FINANCIAL SECRET GENERATION	
25	EXHIBIT 2-24 TBOX	
26	EXHIBIT 2-25 CMEA ALGORITHM	
27	EXHIBIT 2-26 ENHANCED TBOX	
28	EXHIBIT 2-27 ECMEA STRUCTURE	
29	EXHIBIT 2-28 ECMEA TRANSFORMATION AND ITS INVERSE	59
30	EXHIBIT 2-29 ECMEA ALGORITHM HEADER	
31	EXHIBIT 2-30 ECMEA ENCRYPTION/DECRYPTION ALGORITHM FOR THE MOBILE STATI	
32	EXHIBIT 2-31 WRE HEADER	
33	EXHIBIT 2-32 CAVE INITIAL LOADING FOR WIKEY GENERATION	
34	EXHIBIT 2-33 GENERATION OF WIKEY	
35	EXHIBIT 2-34 CODE FOR WIKEY GENERATION	68
36	EXHIBIT 2-35 CAVE INITIAL LOADING FOR WIKEY UPDATE	
37	EXHIBIT 2-36 GENERATION OF WIKEY NEW	
38	EXHIBIT 2-37 CODE FOR WIKEY NEW GENERATION	
39	EXHIBIT 2-38 CAVE INITIAL LOADING FOR WIRELINE INTERFACE AUTHENTICATION	
40	SIGNATURES	73
41	EXHIBIT 2-39 CALCULATION OF AUTH SIGNATURE	
42	EXHIBIT 2-40 CODE FOR CALCULATION OF AUTH SIGNATURE	75
43	EXHIBIT 2-41 CAVE INITIAL LOADING FOR RESIDENTIAL WIRELESS EXTENSION	
44	AUTHENTICATION SIGNATURE	76
45	EXHIBIT 2-42 CALCULATION OF AUTH SIGNATURE	77
46	EXHIBIT 2-43 CODE FOR CALCULATION OF AUTH_SIGNATURE	78
47	EXHIBIT 2-44 GALOIS SHIFT REGISTERS	81
48	EXHIBIT 2-45 HEADER FOR BASIC DATA ENCRYPTION.	83
49	EXHIBIT 2-46 DATAKEY GENERATION	
50	EXHIBIT 2-47 LTABLE GENERATION	
51	EXHIBIT 2-48 DATA ENCRYPTION MASK GENERATION	
52	EXHIBIT 2-49 SCEMA DTC KEY GENERATION	
53	EXHIBIT 2-50 SCEMA DCCH KEY GENERATION	
54	EXHIBIT 2-51 GENERATION OF SCEMA SECRETS	
55	EVHIBIT 2.52 SCEMA SECRET GENERATION	،رر 08

1	EXHIBIT 2-53 SCEMA HEADER FILE	100
2	EXHIBIT 2-54 SCEMA WITH SUBTENDING FUNCTIONS STBOX AND SCEMA_TRANSFORM	105
3	EXHIBIT 2-55 SCEMA KSG FOR VOICE AND MESSAGE CONTENT	112
4	EXHIBIT 2-56 LONG BLOCK ENCRYPTOR FOR VOICE AND MESSAGE CONTENT	114
5	EXHIBIT 2-57 SHORT BLOCK ENCRYPTOR FOR VOICE AND MESSAGE CONTENT	117
6	EXHIBIT 2-58 ENHANCED VOICE PRIVACY	123
7	EXHIBIT 2-59 ENHANCED MESSAGE ENCRYPTION	129
8	EXHIBIT 2-60 ENHANCED DATA MASK GENERATION	134

1. Introduction

This document describes detailed cryptographic procedures for wireless 2 system applications. These procedures are used to perform the security 3 services of mobile station authentication, subscriber message encryption, and encryption key and subscriber voice privacy key 5 generation within wireless equipment. 6 This document is organized as follows: 7 §2 describes the Cellular Authentication, Voice Privacy and Encryption 8 (CAVE) algorithm used for authentication of mobile subscriber equipment and for generation of cryptovariables to be used in other 10 procedures. 11 §2.2 describes the procedure to verify the manual entry of the 12 subscriber authentication key (A-key). 13 describes the generation of intermediate 14 cryptovariables, Shared Secret Data (SSD), from the unique and private 15 subscriber A-key. 16 §2.4 describes the authentication signature calculation procedure. 17 §2.5 describes the procedures used for generating cryptographic keys. 18 These keys include the Voice Privacy Mask (VPM), the Cellular 19 Message Encryption Algorithm (CMEA) key, and Enhanced Cellular 20 Message Encryption Algorithm (ECMEA) secrets and keys. The VPM 21 22 is used to provide forward link and reverse link voice confidentiality over the air interface. The CMEA key is used with the CMEA 23 algorithm for protection of digital data exchanged between the mobile 24 station and the base station. The ECMEA secrets and keys are used with 25 the ECMEA algorithm for enhanced protection of signaling messages. 26 §2.6 describes the Cellular Message Encryption Algorithm (CMEA) 27 and the Enhanced Cellular Message Encryption Algorithm (ECMEA), used for enciphering and deciphering subscriber data exchanged 29 between the mobile station and the base station. 30 §2.7 describes the procedures for key and authentication signature 31 generation for wireless residential extension applications. §2.8 describes the ORYX algorithm and procedures for key and mask 33 generation for encryption and decryption in wireless data services. 34 §2.9 describes the SCEMA algorithm, which may be used for voice and data privacy. 36 §3 provides test data (vectors) that may be employed to verify the 37 correct operation of the cryptographic algorithms described in this 38 document.

Manufacturers are cautioned that no mechanisms should be provided for the display at the ACRE, PB or mobile station (or any other equipment that may be interfaced with it) of valid A-key, SSD_A, SSD_B, MANUFACT_KEY, WIKEY, WRE_KEY or other cryptovariables associated with the cryptographic functions described in this document. The invocation of test mode in the ACRE, PB or mobile station must not alter the operational values of A-key, SSD_A, SSD_B MANUFACT_KEY, WIKEY, WRE_KEY or other cryptovariables.

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.

For example, Exhibit 2-1 shows the mixing registers R[00] through R[15] and the linear feedback shift register (LFSR). In this exhibit, the mixing registers are loaded from left (most significant bit) to right (least significant bit). Similarly, the LFSR is loaded with the most significant bits in its leftmost octet (LFSR A7-A0) and the least significant bits into its rightmost octet (LFSR D7-D0).

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

32	AAV	Authentication Algorithm Version, an 8-bit constant equal to
33		hexadecimal 0xC7, used in the CAVE algorithm. Use of different
34		values for this constant in some future version would allow other
35		"versions" or "flavors" of the basic CAVE algorithm.
36	ACRE	Authorization and Call Routing Equipment. A network device which
37		authorizes the Personal Base and provides automatic call routing.
38	ACRE_PHONE_NUMBER	A 24-bit pattern comprised of the last 6 digits of the ACRE's directory
39		number.

1 2 3 4 5 6	A-key	A 64-bit cryptographic key variable stored in the semi-permanent memory of the mobile station and also known to the Authentication Center (AC or HLR/AC) of the wireless system. It is entered when the mobile station is first put into service with a particular subscriber, and usually will remain unchanged unless the operator determines that its value has been compromised. The A-key is used in the SSD generation procedure.
8	AND	Bitwise logical AND function.
9	Boolean	Describes a quantity whose value is either TRUE or FALSE.
10	CAVE	Cellular Authentication and Voice Encryption algorithm.
11 12	CaveTable	A lookup table consisting of 256 8-bit quantities. The table, partitioned into table0 and table1, is used in the CAVE algorithm.
13	CMEA	Cellular Message Encryption Algorithm.
14 15 16 17	CMEAKEY	A 64-bit cryptographic key stored in eight 8-bit registers identified separately as k0, k1, k7 or CMEAKEY[0 through 7]. The data in these registers results from the action of the CAVE algorithm and is used to encrypt certain messages.
18 19	DataKey	A 32-bit cryptographic key used for generation of masks for encryption and decryption in wireless data services.
20 21	Data_type	A one-bit value indicating whether the financial or non-financial data encryption parameters are used.
22	Directory Number	The telephone network address.
23	ECMEA	Enhanced Cellular Message Encryption Algorithm.
24 25 26 27	ECMEA_KEY	A 64-bit cryptographic key stored in eight 8-bit registers identified separately as ecmea_key[0 through 7]. The data in these registers results from the action of the CAVE algorithm and is used to encrypt financial messages.
28 29 30 31	ECMEA_NF_KEY	A 64-bit cryptographic key stored in eight 8-bit registers identified separately as ecmea_nf_key[0 through 7]. The data in these registers results from the action of the CAVE algorithm and is used to encrypt non-financial messages.
32	ESN	The 32-bit electronic serial number of the mobile station.
33 34	Internal Stored Data	Stored data that is defined locally within the cryptographic procedures and is not accessible for examination or use outside those procedures.
35 36 37	Iteration	Multi-round execution of the CAVE algorithm. All applications of CAVE throughout this document use either four or eight rounds per iteration.
38	k0,k1k7	Eight 8-bit registers whose contents constitute the CMEA key.
39 40	LFSR	A 32-bit Linear Feedback Shift Register used in the CAVE algorithm, which is composed of four 8-bit registers.
41	LFSR_A	The A register, a synonym for bits 31-24 of the LFSR.
42	LFSR_B	The B register, a synonym for bits 23-16 of the LFSR.
43	LFSR_C	The C register, a synonym for bits 15-8 of the LFSR.
44	LFSR_D	The D register, a synonym for bits 7-0 of the LFSR.
45	LFSR-Cycle	An LFSR-cycle consists of the following steps:

1 2 3		 Compute the value of bit A7 using the formula A7 = B6 XOR D2 XOR D1 XOR D0. Save this value temporarily without changing the prior value of the A7 bit in the A register.
4 5		2. Perform a linked 1-bit right shift on the 32-bit LFSR, and discard the D0 bit which has been shifted out.
6 7		3. Use the previously computed and stored value of bit A7 from the first of these three statements.
8	LSB	Least Significant Bit.
9	MSB	Most Significant Bit.
10	OR	Bitwise logical inclusive OR function.
11 12 13	Offset1	An 8-bit quantity that points to one of the 256 4-bit values in table0. Arithmetic operations on Offset1 are performed modulo 256. Also called offset_1.
14 15 16	Offset2	An 8-bit quantity that points to one of the 256 4-bit values in table1. Arithmetic operations on Offset2 are performed modulo 256. Also called offset_2.
17 18 19	offset_key	A 32-bit cryptographic key stored in four 8-bit registers identified separately as offset_key[0 through 3] whose contents are used to create offsets that are passed to ECMEA.
20 21 22 23	offset_nf_key	A 32-bit cryptographic key stored in four 8-bit registers identified separately as offset_nf_key[0 through 3] whose contents are used to create offsets that are passed to ECMEA for use in encryption of non-financial data.
20		illument data.
24 25	PB	Personal Base. A fixed device which provides cordless like service to a mobile station.
24	PB PBID	Personal Base. A fixed device which provides cordless like service to a
24 25		Personal Base. A fixed device which provides cordless like service to a mobile station.
24 25 26	PBID	Personal Base. A fixed device which provides cordless like service to a mobile station. Personal Base Identification Code.
24 25 26 27	PBID RAND_ACRE	Personal Base. A fixed device which provides cordless like service to a mobile station. Personal Base Identification Code. A 32-bit random number which is generated by the PB.
24 25 26 27 28	PBID RAND_ACRE RAND_PB	Personal Base. A fixed device which provides cordless like service to a mobile station. Personal Base Identification Code. A 32-bit random number which is generated by the PB. A 32-bit random number which is generated by the ACRE.
24 25 26 27 28 29	PBID RAND_ACRE RAND_PB RAND_WIKEY	Personal Base. A fixed device which provides cordless like service to a mobile station. Personal Base Identification Code. A 32-bit random number which is generated by the PB. A 32-bit random number which is generated by the ACRE. A 56-bit random number which is generated by the ACRE.
24 25 26 27 28 29	PBID RAND_ACRE RAND_PB RAND_WIKEY RAND_WRE	Personal Base. A fixed device which provides cordless like service to a mobile station. Personal Base Identification Code. A 32-bit random number which is generated by the PB. A 32-bit random number which is generated by the ACRE. A 56-bit random number which is generated by the ACRE. A 19-bit random number which is generated by the PB.
24 25 26 27 28 29 30 31	PBID RAND_ACRE RAND_PB RAND_WIKEY RAND_WRE Round	Personal Base. A fixed device which provides cordless like service to a mobile station. Personal Base Identification Code. A 32-bit random number which is generated by the PB. A 32-bit random number which is generated by the ACRE. A 56-bit random number which is generated by the ACRE. A 19-bit random number which is generated by the PB. A round is one individual execution of the CAVE mixing function. Sixteen separate 8-bit mixing registers used in the CAVE algorithm.
24 25 26 27 28 29 30 31 32 33 34 35	PBID RAND_ACRE RAND_PB RAND_WIKEY RAND_WRE Round R00-R15	Personal Base. A fixed device which provides cordless like service to a mobile station. Personal Base Identification Code. A 32-bit random number which is generated by the PB. A 32-bit random number which is generated by the ACRE. A 56-bit random number which is generated by the ACRE. A 19-bit random number which is generated by the PB. A round is one individual execution of the CAVE mixing function. Sixteen separate 8-bit mixing registers used in the CAVE algorithm. Also called register[0 through 15]. Five 8-bit registers whose content constitutes the 40-bit binary quantity generated after the CMEA key and used to initialize the CAVE
24 25 26 27 28 29 30 31 32 33 34 35 36	PBID RAND_ACRE RAND_PB RAND_WIKEY RAND_WRE Round R00-R15 SEED_NF_KEY	Personal Base. A fixed device which provides cordless like service to a mobile station. Personal Base Identification Code. A 32-bit random number which is generated by the PB. A 32-bit random number which is generated by the ACRE. A 56-bit random number which is generated by the ACRE. A 19-bit random number which is generated by the PB. A round is one individual execution of the CAVE mixing function. Sixteen separate 8-bit mixing registers used in the CAVE algorithm. Also called register[0 through 15]. Five 8-bit registers whose content constitutes the 40-bit binary quantity generated after the CMEA key and used to initialize the CAVE algorithm for generation of the ECMEA_NF key and offset_nf keys. SSD is an abbreviation for Shared Secret Data. It consists of two

1 2 3 4	SSD_B	A 64-bit binary quantity in the semi-permanent memory of the mobile station and also known to the Authentication Center. It may be shared with the serving MSC. It is used in the computation of the CMEA key, VPM and DataKey.
5 6	SSD_B_NEW	The revised 64-bit quantity held separately from SSD_B, generated as a result of the SSD generation process.
7 8	Sync	A 16-bit value provided by the air interface used to generate offsets for ECMEA.
9 10	table0	The low-order four bits of the 256-octet lookup table used in the CAVE algorithm. Computed as CaveTable[] AND $0x0F$.
11 12	table1	The high-order four bits of the 256-octet lookup table used in the CAVE algorithm. Computed as CaveTable[] AND 0xF0.
13 14 15	VPM	Voice Privacy Mask. This name describes a 520-bit entity that may be used for voice privacy functions as specified in wireless system standards.
16 17	WIKEY	Wireline Interface key. A 64-bit pattern stored in the PB and the ACRE (in semi-permanent memory).
18 19	WIKEY_NEW	A 64-bit pattern stored in the PB and the ACRE. It contains the value of an updated WIKEY.
20 21	WRE_KEY	Wireless Residential Extension key. A 64-bit pattern stored in the PB and the MS (in semi-permanent memory).
22	XOR	Bitwise logical exclusive OR function.

2. Procedures

2.1. **CAVE**

CAVE is a software-compatible non-linear mixing function shown in Exhibit 2-1. Its primary components are a 32-bit linear-feedback shift register (LFSR), sixteen 8-bit mixing registers, and a 256-entry lookup table. The table is organized as two (256 x 4 bit) tables. The 256-octet table is listed in Exhibit 2-5. The low order four bits of the entries comprise *table0* and the high order four bits of the entries comprise *table1*.

The pictorial arrangement of Exhibit 2-1 shows that the linear-feedback shift register (LFSR) consists of the 8-bit register stages A, B, C, and D. The CAVE process repeatedly uses the LFSR and table to randomize the contents of the 8-bit mixing register stages R00, R01, R02, R03, R04, R05, R06, R07, R08, R09, R10, R11, R12, R13, R14, and R15. Two lookup table pointer offsets further randomize table access. The registers are shifted one bit to the right. Finally, eight 16-entry permutation recipes are embedded in the lookup tables to "shuffle" registers R00 through R15 after each computational "round" through the algorithm.

The algorithm operation consists of three steps: an initial loading, a repeated randomization consisting of four or eight "rounds", and processing of the output. Initial loading consists of filling the LFSR, register stages R00 through R15, and the pointer offsets with information that is specific to the application. The randomization process is common to all cases that will be described in the later sections. Randomization is a detailed operation; it is described below by means of Exhibit 2-1, Exhibit 2-2, and Exhibit 2-5. The output processing utilizes the final (randomized) contents of R00 through R15 in a simple function whose result is returned to the calling process.

The CAVE Algorithm may be applied in a number of different cases. In each, there are different initialization requirements, and different output processing. All cases are detailed in §2.2 through §2.9 of this document.

Exhibit 2-1 CAVE Elements

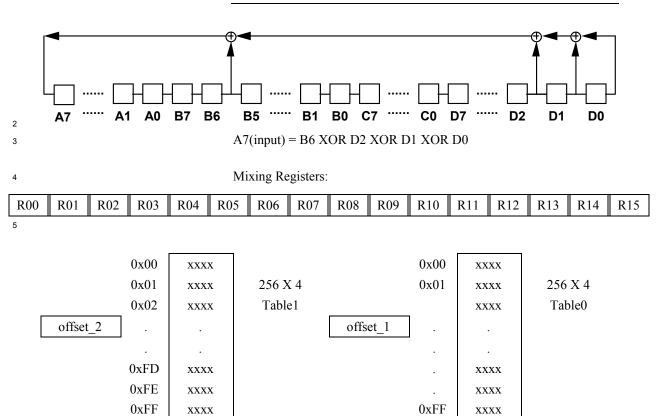


Exhibit 2-2 CAVE Algorithm External Header

```
#ifndef CAVE H
2
    #define CAVE H
3
    /* external header for CAVE and related procedures */
6
    /* function declarations */
8
9
    void CAVE (const int number of rounds,
10
               int *offset_1,
               int *offset_2);
11
12
    void A_Key_Checksum(const char A KEY DIGITS[20],
13
                          char A KEY CHECKSUM[6]);
14
15
    int A Key Verify(const char A KEY DIGITS[26]);
16
17
    void SSD Generation(const unsigned char RANDSSD[7]);
18
19
20
    void SSD Update(void);
21
    unsigned long Auth Signature (const unsigned char RAND CHALLENGE [4],
22
                                   const unsigned char AUTH DATA[3],
23
                                   const unsigned char *SSD_AUTH,
24
                                   const int SAVE REGISTERS);
25
26
    void Key VPM Generation(void);
27
28
    void CMEA(unsigned char *msg buf, const int octet count);
29
30
31
    /* global variable definitions */
32
33
    #ifdef CAVE_SOURCE_FILE
34
    #define CAVE GLOBAL
35
    #else
    #define CAVE GLOBAL extern
36
    #endif
37
38
    /* externally available results */
39
40
    CAVE GLOBAL
41
42
    unsigned char
                       cmeakey[8];
43
    CAVE GLOBAL
44
45
    unsigned char
                       VPM[65];
46
    CAVE GLOBAL
47
48
    unsigned char
                       SAVED LFSR[4];
    CAVE GLOBAL
49
                       SAVED OFFSET 1;
    int
50
    CAVE GLOBAL
51
                       SAVED OFFSET_2;
    int
52
    CAVE GLOBAL
53
54
    unsigned char
                       SAVED RAND[4];
55
    CAVE GLOBAL
56
    unsigned char
                       SAVED DATA[3];
57
```

```
/* global constant definitions */
1
   #ifndef CAVE_SOURCE_FILE
3
  CAVE_GLOBAL
  unsigned char CaveTable[256];
6
  CAVE GLOBAL
8
   unsigned char ibox[256];
9
10
  #endif // ifndef CAVE SOURCE FILE
11
12
   #endif // ifndef CAVE H
13
14
   15
16
17
```

Exhibit 2-3 CAVE Algorithm Internal Header

```
/* internal header for CAVE, used by all cryptographic source files */
2
3
    #include "cave.h" /* see Exhibit 2-2 */
4
    /* authentication algorithm version (fixed) */
6
8
    #define AAV 0xC7
10
    #define LOMASK
                        0x0F
    #define HIMASK
                         0xF0
11
    #define TRUE
12
    #define FALSE
13
14
15
    /* NAM stored data */
16
17
    extern
   unsigned char ESN[4];
18
19
20
    extern
21
   unsigned char A key[8];
22
23
   extern
   unsigned char
                    SSD_A_NEW[8], SSD_A[8];
24
25
26
   extern
   unsigned char
                      SSD B NEW[8], SSD B[8];
27
28
    /* saved outputs */
29
30
31
    CAVE_GLOBAL
    unsigned char
32
                    LFSR[4];
33
34
    #define LFSR_A LFSR[0]
    #define LFSR_B LFSR[1]
35
    #define LFSR_C LFSR[2]
36
    #define LFSR D LFSR[3]
37
38
    CAVE GLOBAL
39
    unsigned char Register[16];
40
41
```

Exhibit 2-4 CAVE Algorithm

```
#define CAVE SOURCE FILE
2
    #include "cavei.h" /* see Exhibit 2-3 */
3
4
    5
6
7
    /* table0 is the 4 lsbs of the array,
8
       table1 is the 4 msbs of the array */
9
10
    unsigned char
                      CaveTable[256] =
11
          0xd9, 0x23, 0x5f, 0xe6, 0xca, 0x68, 0x97, 0xb0,
12
          0x7b, 0xf2, 0x0c, 0x34, 0x11, 0xa5, 0x8d, 0x4e,
13
          0x0a, 0x46, 0x77, 0x8d, 0x10, 0x9f, 0x5e, 0x62,
14
          0xf1, 0x34, 0xec, 0xa5, 0xc9, 0xb3, 0xd8, 0x2b,
15
          0x59, 0x47, 0xe3, 0xd2, 0xff, 0xae, 0x64, 0xca,
16
          0x15, 0x8b, 0x7d, 0x38, 0x21, 0xbc, 0x96, 0x00,
17
          0x49, 0x56, 0x23, 0x15, 0x97, 0xe4, 0xcb, 0x6f,
18
          0xf2, 0x70, 0x3c, 0x88, 0xba, 0xd1, 0x0d, 0xae,
19
20
          0xe2, 0x38, 0xba, 0x44, 0x9f, 0x83, 0x5d, 0x1c,
          0xde, 0xab, 0xc7, 0x65, 0xf1, 0x76, 0x09, 0x20,
21
          0x86, 0xbd, 0x0a, 0xf1, 0x3c, 0xa7, 0x29, 0x93,
22
          0xcb, 0x45, 0x5f, 0xe8, 0x10, 0x74, 0x62, 0xde,
23
          0xb8, 0x77, 0x80, 0xd1, 0x12, 0x26, 0xac, 0x6d,
24
          0xe9, 0xcf, 0xf3, 0x54, 0x3a, 0x0b, 0x95, 0x4e,
25
          0xb1, 0x30, 0xa4, 0x96, 0xf8, 0x57, 0x49, 0x8e,
26
          0x05, 0x1f, 0x62, 0x7c, 0xc3, 0x2b, 0xda, 0xed,
27
          0xbb, 0x86, 0x0d, 0x7a, 0x97, 0x13, 0x6c, 0x4e,
28
          0x51, 0x30, 0xe5, 0xf2, 0x2f, 0xd8, 0xc4, 0xa9,
29
          0x91, 0x76, 0xf0, 0x17, 0x43, 0x38, 0x29, 0x84,
30
          0xa2, 0xdb, 0xef, 0x65, 0x5e, 0xca, 0x0d, 0xbc,
31
          0xe7, 0xfa, 0xd8, 0x81, 0x6f, 0x00, 0x14, 0x42,
32
33
          0x25, 0x7c, 0x5d, 0xc9, 0x9e, 0xb6, 0x33, 0xab,
34
          0x5a, 0x6f, 0x9b, 0xd9, 0xfe, 0x71, 0x44, 0xc5,
          0x37, 0xa2, 0x88, 0x2d, 0x00, 0xb6, 0x13, 0xec,
35
          0x4e, 0x96, 0xa8, 0x5a, 0xb5, 0xd7, 0xc3, 0x8d,
36
          0x3f, 0xf2, 0xec, 0x04, 0x60, 0x71, 0x1b, 0x29,
37
          0x04, 0x79, 0xe3, 0xc7, 0x1b, 0x66, 0x81, 0x4a,
38
          0x25, 0x9d, 0xdc, 0x5f, 0x3e, 0xb0, 0xf8, 0xa2,
39
          0x91, 0x34, 0xf6, 0x5c, 0x67, 0x89, 0x73, 0x05,
40
          0x22, 0xaa, 0xcb, 0xee, 0xbf, 0x18, 0xd0, 0x4d,
41
          0xf5, 0x36, 0xae, 0x01, 0x2f, 0x94, 0xc3, 0x49,
42
43
          0x8b, 0xbd, 0x58, 0x12, 0xe0, 0x77, 0x6c, 0xda
```

```
unsigned char ibox[256] =
          0xdd, 0xf3, 0xf7, 0x90, 0x0b, 0xf5, 0x1a, 0x48,
3
          0x20, 0x3c, 0x84, 0x04, 0x19, 0x16, 0x22, 0x47,
          0x6d, 0xa8, 0x8e, 0xc8, 0x9f, 0x8d, 0x0d, 0xb5,
          0xc2, 0x0c, 0x06, 0x2f, 0x43, 0x60, 0xf0, 0xa4,
          0x08, 0x99, 0x0e, 0x36, 0x98, 0x3d, 0x2e, 0x81,
          0xcb, 0xab, 0x5c, 0xd5, 0x3f, 0xee, 0x26, 0x1b,
          0x94, 0xd9, 0xfc, 0x68, 0xde, 0xcd, 0x23, 0xed,
9
          0x96, 0xc5, 0xdc, 0x45, 0x09, 0x25, 0x4f, 0x2c,
10
          0x62, 0x53, 0xbf, 0x1c, 0x95, 0x3b, 0x89, 0x0f,
11
          0x07, 0x56, 0x7f, 0xbd, 0xaa, 0xb7, 0xff, 0x3e,
12
          0x86, 0x77, 0x54, 0x41, 0x52, 0xd4, 0x49, 0xb8,
13
          0xc7, 0x9e, 0x82, 0x71, 0x2a, 0xd0, 0x78, 0x9c,
14
          0x1d, 0x6a, 0x40, 0xae, 0xf4, 0xaf, 0xf2, 0xe9,
15
          0x33, 0x80, 0x61, 0xb4, 0xc0, 0x10, 0xa7, 0xbb,
16
          0xb6, 0x5b, 0x73, 0x72, 0x79, 0x7c, 0x8c, 0x51,
17
          0x5e, 0x74, 0xfb, 0xe6, 0x75, 0xd6, 0xef, 0x4a,
18
          0x69, 0x27, 0x5a, 0xb3, 0x0a, 0xe8, 0x50, 0xa0,
19
          0xca, 0x46, 0xc3, 0xea, 0x76, 0x15, 0x12, 0xc6,
20
          0x03, 0x97, 0xa3, 0xd1, 0x30, 0x44, 0x38, 0x91,
21
          0x24, 0x21, 0xc1, 0xdb, 0x5f, 0xe3, 0x59, 0x14,
22
          0x87, 0xa2, 0xa1, 0x92, 0x1f, 0xe2, 0xbc, 0x6e,
23
          0x11, 0xbe, 0x4c, 0x29, 0xe4, 0xc9, 0x63, 0x65,
24
          0xcc, 0xfa, 0xf1, 0x83, 0x6b, 0x17, 0x70, 0x4d,
25
          0x57, 0xd3, 0xfe, 0x6f, 0xa6, 0x4b, 0xa9, 0x42,
26
          0x6c, 0x9a, 0x18, 0x8a, 0xd2, 0x39, 0x8f, 0x58,
27
          0x13, 0xad, 0x88, 0x28, 0xb0, 0x35, 0xd7, 0xe1,
28
          0x5d, 0x93, 0xc4, 0xb9, 0x55, 0x2b, 0x7d, 0xce,
29
          0xe0, 0x31, 0xfd, 0x9b, 0x3a, 0x00, 0x34, 0xe5,
30
          0xd8, 0xcf, 0xa5, 0x9d, 0xac, 0xdf, 0x7b, 0xf9,
31
          0x85, 0x67, 0x8b, 0xf6, 0xf8, 0x37, 0x2d, 0x7e,
32
          0x1e, 0xb2, 0x66, 0x01, 0x64, 0x05, 0xeb, 0x02,
33
          0xec, 0xe7, 0xb1, 0x7a, 0x32, 0xda, 0xba, 0x4e
34
35
```

```
/* CAVE local functions */
1
2
    static unsigned char bit val(const unsigned char octet, const int bit)
3
5
        return((octet << (7 - bit)) & 0x80);
6
    static void LFSR cycle(void)
8
9
       unsigned char temp;
10
        int i;
11
12
        temp = bit val(LFSR B,6);
13
        temp ^= bit val(LFSR_D,2);
14
        temp ^= bit_val(LFSR_D,1);
15
       temp ^= bit_val(LFSR_D,0);
16
17
       /* Shift right LFSR, Discard LFSR D[0] bit */
18
19
        for (i = 3; i > 0; i--)
20
21
           LFSR[i] >>= 1;
22
           if (LFSR[i-1] & 0x01)
23
              LFSR[i] \mid = 0x80;
24
25
26
       LFSR[0] >>= 1;
27
       LFSR A |= temp;
28
29
30
    static void Rotate_right_registers(void)
31
32
       unsigned int temp reg;
33
        int i;
34
35
        temp reg = Register[15]; /* save lsb */
36
37
        for (i = 15; i > 0; i--)
38
39
           Register[i] >>= 1;
40
           if (Register[i-1] & 0x01)
41
              Register[i] = 0x80;
42
43
44
       Register[0] >>= 1;
45
        if (temp reg & 0x01)
46
           Register[0] = 0x80;
47
48
49
50
51
```

```
void CAVE(const int number of rounds,
1
               int *offset 1,
2
               int *offset 2)
3
       unsigned char
                            temp_reg0;
5
       unsigned char
                           lowNibble;
6
       unsigned char
                           hiNibble;
7
       unsigned char
                           temp;
8
                            round_index;
       int
9
       int
                            R index;
10
                            fail count;
       int
11
       unsigned char
                            T[16];
12
13
        for (round_index = number_of_rounds - 1;
14
             round_index >= 0;
15
             round_index--)
16
17
           /* save R0 for reuse later */
18
           temp reg0 = Register[0];
19
20
           for (R index = 0; R index < 16; R index++)</pre>
21
22
              fail count = 0;
23
              while (1)
24
25
                  *offset_1 += (LFSR_A ^ Register[R_index]);
26
                  /* will overflow; mask to prevent */
27
                  *offset 1 &= 0xff;
28
                  lowNibble = CaveTable[*offset 1] & LOMASK;
29
                  if (lowNibble == (Register[R index] & LOMASK))
30
31
                     LFSR cycle();
32
                     fail count++;
33
                     if (\overline{f}ail\ count == 32)
34
35
                        LFSR D++; /* no carry to LFSR C */
36
                        break;
37
38
39
                  else break;
40
41
42
```

```
fail_count = 0;
1
              while (1)
3
                  *offset_2 += (LFSR_B ^ Register[R_index]);
                  /* will overflow; mask to prevent */
                  *offset 2 &= 0xff;
6
                  hiNibble = CaveTable[*offset 2] & HIMASK;
                  if (hiNibble == (Register[R index] & HIMASK))
8
9
                     LFSR cycle();
10
                     fail count++;
11
                     if (\overline{fail} count == 32)
12
13
                        LFSR D++; /* no carry to LFSR C */
14
                        break;
15
16
                  }
17
                  else
18
                     break;
19
20
21
22
              temp = lowNibble | hiNibble;
23
              if (R index == 15)
24
                  Register[R index] = temp reg0 ^ temp;
25
26
                  Register[R_index] = Register[R_index+1] ^ temp;
27
28
              LFSR_cycle();
29
           }
30
31
           Rotate right registers();
32
33
           /* shuffle the mixing registers */
34
           for (R index = 0; R index < 16; R index++)
35
36
              temp = CaveTable[16*round index + R index] & LOMASK;
37
              T[temp] = Register[R index];
38
39
           for (R_index = 0; R_index < 16; R_index++)</pre>
40
              Register[R_index] = T[R_index];
41
42
43
44
```

Exhibit 2-5 CAVE Table

table0 is comprised by the 4 LSBs of the array table1 is comprised by the 4 MSBs of the array

This table is read by rows, e.g. CaveTable[0x12] = 0x77.

hi/lo	0	1	2	3	4	5	6	7	8	9	A	В	С	D	Е	F
0	D9	23	5F	E6	CA	68	97	В0	7B	F2	0C	34	11	A5	8D	4E
1	0A	46	77	8D	10	9F	5E	62	F1	34	EC	A5	C9	В3	D8	2B
2	59	47	E3	D2	FF	AE	64	CA	15	8B	7D	38	21	BC	96	00
3	49	56	23	15	97	E4	СВ	6F	F2	70	3C	88	BA	D1	0D	AE
4	E2	38	BA	44	9F	83	5D	1C	DE	AB	C7	65	F1	76	09	20
5	86	BD	0A	F1	3C	A7	29	93	СВ	45	5F	E8	10	74	62	DE
6	В8	77	80	D1	12	26	AC	6D	E9	CF	F3	54	3A	0B	95	4E
7	B1	30	A4	96	F8	57	49	8E	05	1F	62	7C	C3	2B	DA	ED
8	BB	86	0D	7A	97	13	6C	4E	51	30	E5	F2	2F	D8	C4	A9
9	91	76	F0	17	43	38	29	84	A2	DB	EF	65	5E	CA	0D	BC
A	E7	FA	D8	81	6F	00	14	42	25	7C	5D	C9	9E	В6	33	AB
В	5A	6F	9B	D9	FE	71	44	C5	37	A2	88	2D	00	В6	13	EC
C	4E	96	A8	5A	B5	D7	C3	8D	3F	F2	EC	04	60	71	1B	29
D	04	79	E3	C7	1B	66	81	4A	25	9D	DC	5F	3E	В0	F8	A2
Е	91	34	F6	5C	67	89	73	05	22	AA	СВ	EE	BF	18	D0	4D
F	F5	36	AE	01	2F	94	C3	49	8B	BD	58	12	E0	77	6C	DA

2.2. Authentication Key (A-Key) Procedures

2.2.1. A-Key Checksum Calculation

3	Procedure name:
4	A_Key_Checksum
5	Inputs from calling process:
6	A KEY DIGITS 20 decimal digits
7	ESN 32 bits
8	Inputs from internal stored data:
9	AAV 8 bits
10	Outputs to calling process:
11	A_KEY_CHECKSUM 6 decimal digits
12	Outputs to internal stored data:
13	None.
14	This procedure computes the checksum for an A-key to be entered into
15	a mobile station. In a case where the number of digits to be entered is
16	less than 20, the leading most significant digits will be set equal to zero.
17	The generation of the A-key is the responsibility of the service
18	provider. A-keys should be chosen and managed using procedures that
19	minimize the likelihood of compromise.
20	The checksum provides a check for the accuracy of the A-Key when
21	entered into a mobile station. The 20 A-Key digits are converted into a
22	64-bit representation to serve as an input to CAVE, along with the
23	mobile station's ESN. CAVE is then run in the same manner as for the
24	Auth_Signature procedure, and its 18-bit response is the A-Key
25	checksum. The checksum is returned as 6 decimal digits for entry into
26	the mobile station.
27	The first decimal digit of the A-Key to be entered is considered to be
28	the most significant of the 20 decimal digits, followed in succession by

12345678901234567890

the other nineteen. A decimal to binary conversion process converts the digit sequence into its equivalent mod-2 representation. For example,

have a hexadecimal equivalent of

the 20 digits

30

31

33

34

AB54A98CEB1F0AD2.

15

17

CAVE will be initialized as shown in Exhibit 2-6. First, the 32 most significant bits of the 64-bit entered number will be loaded into the LFSR. If this 32-bit pattern fills the LFSR with all zeros, then the LFSR will be loaded with the ESN. Then, in all instances, the entire 64-bit entered number will be put into R00 through R07. The least significant 24 bits will be repeated into R09, R10, and R11. Authentication Algorithm Version (hexadecimal C7) will occupy R08. and ESN will be loaded into R12 through R15. CAVE will then be performed for eight rounds, as described in §2.1. The checksum is obtained from the final value of CAVE registers R00, R01, R02, R13, R14, and R15. The two most significant bits of the checksum are equal to the two least significant bits of R00 XOR R13. The next eight bits of the checksum are equal to R01 XOR R14. Finally, the least significant bits of the checksum are equal to R02 XOR R15.

The 18-bit checksum is returned as 6 decimal digits for entry into the mobile station.

Exhibit 2-6 CAVE Initial Loading for A-key Checksum

CAVE Element	Source Identifier		Size (Bits)
	32 MSBs of A-key all zeros	32 MSBs of A-key not all zeros	
LFSR	ESN	32 MSBs of A-key	32
Register [0-7]	A-key	A-key	64
Register [8]	AAV	AAV	8
Register [9-11]	24 LSBs of A-key	24 LSBs of A-key	24
Register [12-15]	ESN	ESN	32

Exhibit 2-7 A-key Checksum

```
/* A Key Checksum has the same header as CAVE (see Exhibit 2-4) */
2
3
    static void mul10(unsigned char i64[8], unsigned int carry)
4
5
6
       int i;
7
       unsigned int temp;
8
9
       for (i = 7; i >= 0; i--)
10
           temp = ((unsigned int)(i64[i]) * 10) + carry;
11
           i64[i] = temp \& 0xFF;
12
           carry = temp >> 8;
13
14
    }
15
16
    static unsigned long Calc Checksum(const unsigned char A key[8])
17
18
       int i,offset_1,offset_2;
19
20
       unsigned long A key checksum;
21
       /* see if 32 MSB are zero */
22
23
       if ((A_key[0] | A_key[1] | A_key[2] | A_key[3]) != 0)
24
25
           /* put 32 MSB into LFSR */
26
           for (i = 0; i < 4; i++)
27
              LFSR[i] = A key[i];
28
29
       else
30
31
32
           /* put ESN into LFSR */
33
           for (i = 0; i < 4; i++)
              LFSR[i] = ESN[i];
34
35
36
       /* put A key into r0-r7 */
37
38
       for (i = 0; i < 8; i++)
39
           Register[i] = A key[i];
40
41
       Register[8] = AAV;
42
43
       /* put ls 24 bits of A key into r9-r11 */
44
45
       for (i = 9; i < 12; i++)
46
           Register[i] = A key[5+i-9];
47
48
       /* put ESN into r12-r15 */
49
       for (i = 12; i < 16; i++)
50
          Register[i] = ESN[i-12];
51
52
       offset_1 = offset_2 = 128;
53
       CAVE(8, &offset 1, &offset 2);
54
55
```

```
A_key_checksum =
1
             ( ((unsigned long) (Register[0] ^ Register[13]) << 16) + ((unsigned long) (Register[1] ^ Register[14]) << 8) +
               ((unsigned long)(Register[2] ^ Register[15]))) & 0x3ffff;
        return (A_key_checksum);
6
7
8
     /* A KEY DIGITS contains the ASCII digits in the order to be entered */
9
10
    void A Key Checksum(const char A KEY DIGITS[20],
11
                           char A KEY CHECKSUM[6])
12
13
        int i;
14
        unsigned char temp_A_key[8];
15
        unsigned long A_key_checksum;
16
17
        /* convert digits to 64-bit representation in temp A key */
18
19
        for (i = 0; i < 8; i++)
20
           temp A key[i] = 0;
21
22
        for (i = 0; i < 20; i++)
23
24
           mul10(temp A key, (unsigned int) (A KEY DIGITS[i] - '0'));
25
26
27
        A_key_checksum = Calc_Checksum(temp_A_key);
28
29
        /* convert checksum to decimal digits */
30
31
        for (i = 0; i < 6; i++)
32
33
           A KEY CHECKSUM[5-i] = '0' + (char) (A key checksum % 10);
34
           A key checksum /= 10;
35
36
37
```

2.2.2. A-Key Verification

1

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

36

37

38

2	Procedure name:	
3	A_Key_Verify	
4	Inputs from calling process:	
5	A_KEY_DIGITS	from 6 to 26 decimal digits
6	ESN	32 bits
7	Inputs from internal stored data:	
8	AAV	8 bits
9	Outputs to calling process:	
10	A_KEY_VERIFIED	Boolean
11	Outputs to internal stored data:	
12	A-key	64 bits
13	SSD_A	64 bits (set to zero)
14	SSD_B	64 bits (set to zero)
15		o the mobile station by any of several
16	methods. These include direct e	electronic entry, over-the-air procedures,
17	and manual entry via the mob	oile station's keypad. This procedure

and manual entry via the mobile station's keypad. This procedure verifies the A-key entered into a mobile station via the keypad.

The default value of the A-key when the mobile station is shipped from the factory will be all binary zeros. The value of the A-key is specified by the operator and is to be communicated to the subscriber according to the methods specified by each operator. A multiple NAM mobile station will require multiple A-keys, as well as multiple sets of the corresponding cryptovariables per A-key.

While A-key digits are being entered from a keypad, the mobile station transmitter shall be disabled.

When the A-key digits are entered from a keypad, the number of digits entered is to be at least 6, and may be any number of digits up to and including 26 digits. In a case where the number of digits entered is less than 26, the leading most significant digits will be set equal to zero, in order to produce a 26-digit quantity called the "entry value".

The verification procedure checks the accuracy of the 26 decimal digit entry value. If the verification is successful, the 64-bit pattern determined by the first 20 digits of the entry value will be written to the subscriber's semi-permanent memory as the A-key. Furthermore, the SSD A and the SSD B will be set to zero. The return value A_KEY_VERIFIED will be set to TRUE. In the case of a mismatch, A_KEY_VERIFIED is set to FALSE, and no internal data is updated.

1 2 3 4 5	The first decimal digit of the "entry value" is considered to be the most significant of the 20 decimal digits, followed in succession by the other nineteen. The twenty-first digit is the most significant of the check digits, followed in succession by the remaining five. For example, the 26 digits	
6	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0, 1 3 1 1 3 6	
7	has a hexadecimal equivalent of	
8	AB54A98CEB1F0AD2, 20040.	

Exhibit 2-8 A-key Verification

```
/* A Key Verify has the same header as CAVE (see Exhibit 2-4) */
2
3
    /* A KEY DIGITS contains the ASCII digits in the order entered */
4
5
    int A Key Verify(const char A KEY DIGITS[26])
6
7
8
        int i;
        unsigned char temp_A_key[8];
9
10
       unsigned long entered checksum;
11
        /* convert first 20 digits to 64-bit representation in temp A key */
12
13
        for (i = 0; i < 8; i++)
14
           temp A key[i] = 0;
15
16
        for (i = 0; i < 20; i++)
17
18
           mul10(temp A key, (unsigned int) (A KEY DIGITS[i] - '0'));
19
20
21
        /* convert last 6 digits to entered checksum */
22
23
        entered_checksum = 0;
24
        for (i = 20; i < 26; i++)
25
26
           entered checksum = (entered checksum * 10)
27
              + (A KEY DIGITS[i] - '0');
28
29
30
        if(Calc_Checksum(temp_A_key) == entered_checksum)
31
32
33
           for (i = 0; i < 8; i++)
34
              A_key[i] = temp_A_key[i];
35
              S\overline{SD} A[i] = SSD \overline{B}[\overline{i}] = 0;
36
37
           return TRUE;
38
39
        else
40
41
        {
42
           return FALSE;
43
44
```

2.3. SSD Generation and Update

2.3.1. SSD Generation Procedure

4	Procedure name:		
5	SSD_Generation		
6	Inputs from calling process:		
7	RANDSSD	56 bits	
8	ESN	32 bits	
9	Inputs from internal stored d	ata:	
10	AAV	8 bits	
11	A-key	64 bits	
12	Outputs to calling process:		
13	None.		
14	Outputs to internal stored da	ta:	
15	SSD_A_NEW	64 bits	
16	SSD_B_NEW	64 bits	

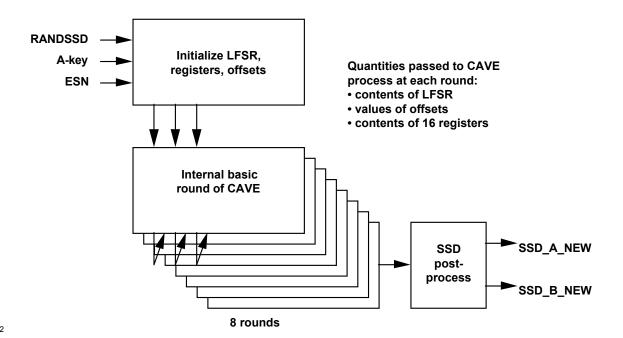
This procedure performs the calculation of Shared Secret Data. The result is held in memory as SSD_A_NEW and SSD_B_NEW until the SSD_Update procedure (§2.3.2) is invoked. Exhibit 2-9 shows the process graphically. Exhibit 2-10 indicates the operations in ANSI C.

The input variables for this procedure are: RANDSSD (56 bits), Authentication Algorithm Version (8 bits), ESN (32 bits), and A-key (64 bits). CAVE will be initialized as follows. First, the LFSR will be loaded with the 32 least significant bits of RANDSSD XOR'd with the 32 most significant bits of A-key XOR'd with the 32 least significant bits of A-key. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be loaded with the 32 least significant bits of RANDSSD to prevent a trivial null result.

Registers R00 through R07 will be initialized with A-key, R08 will be the 8-bit Authentication Algorithm Version (11000111). R09, R10, and R11 will be the most significant bits of RANDSSD, and the ESN will be loaded into R12 through R15. Offset1 and Offset2 will initially be set to 128.

CAVE will be run for 8 rounds as previously described in §2.1. When this is complete, registers R00 through R07 will become SSD_A_NEW and Registers R08 through R15 will become SSD_B_NEW.

Exhibit 2-9 Generation of SSD A NEW and SSD B NEW



Information disclosed in this document is subject to the export jurisdiction of the US Department of Commerce as specified in Export Administration Regulations (title 15 CFR parts 730 through 774 inclusive).

25

Exhibit 2-10 SSD Generation

```
/* SSD Generation has the same header as CAVE (see Exhibit 2-4) */
2
3
4
    void SSD Generation(const unsigned char RANDSSD[7])
5
       int i,offset 1,offset 2;
6
7
8
       for (i = 0; i < 4; i++)
9
10
          LFSR[i] = RANDSSD[i+3] ^A_{key}[i] ^A_{key}[i+4];
11
12
       if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
13
14
          for (i = 0; i < 4; i++)
15
              LFSR[i] = RANDSSD[i+3];
16
17
18
       for (i = 0; i < 8; i++)
19
20
          Register[i] = A_key[i];
21
       Register[8] = AAV;
22
23
       for (i = 9; i < 12; i++)
24
          Register[i] = RANDSSD[i-9];
25
26
       for (i = 12; i < 16; i++)
27
          Register[i] = ESN[i-12];
28
29
       offset_1 = offset_2 = 128;
30
31
       CAVE(8, &offset_1, &offset_2);
32
33
       for (i = 0; i < 8; i++)
34
           SSD_A_NEW[i] = Register[i];
35
           SSD B NEW[i] = Register[i+8];
36
37
38
```

2.3.2. SSD Update Procedure

1

14

15

16

17

18

19

20

21

22

23 24

25

```
Procedure name:
                                               SSD Update
                                         Inputs from calling process:
                                               None.
5
                                         Inputs from internal stored data:
6
                                                                            64 bits
                                               SSD_A_NEW
                                               SSD B NEW
                                                                            64 bits
                                         Outputs to calling process:
9
                                               None.
10
                                         Outputs to internal stored data:
11
                                               SSD A
                                                                            64 bits
12
                                               SSD B
                                                                            64 bits
13
```

This procedure copies the values SSD_A_NEW and SSD_B_NEW into the stored SSD_A and SSD_B. Exhibit 2-11 indicates the operations in ANSI C.

The values SSD_A_NEW and SSD_B_NEW calculated by the SSD_Generation procedure (§2.3.1) should be validated prior to storing them permanently as SSD_A and SSD_B. The base station and the mobile station should exchange validation data sufficient to determine that the values of the Shared Secret Data are the same in both locations. When validation is completed successfully, the SSD_Update procedure is invoked, setting SSD_A to SSD_A_NEW and setting SSD_B to SSD_B_NEW.

Exhibit 2-11 SSD Update

```
/* SSD Update has the same header as CAVE (see Exhibit 2-4) */
26
27
    void SSD Update(void)
28
29
        int i;
30
31
        for (i = 0; i < 8; i++)
32
33
           SSD A[i] = SSD A NEW[i];
34
           SSD B[i] = SSD B NEW[i];
35
36
37
38
```

33

34

36

37

2.4. Authentication Signature Calculation Procedure

	-	
2	Procedure name:	
3	Auth_Signature	
4	Inputs from calling process:	
5	RAND_CHALLENGE 32 bits	
6	ESN 32 bits	
7	AUTH DATA 24 bits	
8	SSD_AUTH 64 bits	
9	SAVE_REGISTERS Boolean	
10	Inputs from internal stored data:	
11	AAV 8 bits	
12	Outputs to calling process:	
13	AUTH_SIGNATURE 18 bits	
14	Outputs to internal stored data:	
15	SAVED LFSR 32 bits	
16	SAVED_OFFSET_1 8 bits	
17	SAVED_OFFSET_2 8 bits	
18	SAVED_RAND 32 bits	
19	SAVED_DATA 24 bits	
20	This procedure is used to calculate 18-bit signatures used for verifying	
21	the authenticity of messages used to request wireless system services,	
22	and for verifying Shared Secret Data.	
23	The initial loading of CAVE for calculation of authentication signatures	
24	is given in Exhibit 2-12.	
25	AAV is as defined in §1.1.	
26	For authentication of mobile station messages and for base station	
27	challenges of a mobile station, RAND_CHALLENGE should be	
28	selected by the authenticating entity (normally the HLR or VLR).	
29	RAND_CHALLENGE must be received by the mobile station	

For authentication of mobile station messages and for base station challenges of a mobile station, RAND_CHALLENGE should be selected by the authenticating entity (normally the HLR or VLR). RAND_CHALLENGE must be received by the mobile station executing this procedure. Results returned by the mobile station should include check data that can be used to verify that the RAND_CHALLENGE value used by the mobile station matches that used by the authenticating entity.

For mobile station challenges of a base station, as performed during the verification of Shared Secret Data, the mobile station should select RAND_CHALLENGE. The selected value of RAND_CHALLENGE must be received by the base station executing this procedure.

When this procedure is used to generate an authentication signature for a message, AUTH_DATA should include a part of the message to be authenticated. The contents should be chosen to minimize the possibility that other messages would produce the same authentication signature.

SSD_AUTH should be either SSD_A or SSD_A_NEW computed by the SSD_Generation procedure, or SSD_A as obtained from the HLR/AC.

Exhibit 2-12 CAVE Initial Loading for Authentication Signatures

CAVE Item	Source Identifier	Size (Bits)
LFSR	RAND_CHALLENGE	32
Reg [0-7]	SSD_AUTH	64
Reg [8]	AAV	8
Reg [9-11]	AUTH_DATA	24
Reg [12-15]	ESN	32

CAVE is run for eight rounds. The 18-bit result is AUTH_SIGNATURE. Exhibit 2-13 shows the process in graphical form, while ANSI C for the process is given in Exhibit 2-14.

The LFSR will initially be loaded with RAND_CHALLENGE. This value will be XOR'd with the 32 most significant bits of SSD_AUTH XOR'd with the 32 least significant bits of SSD_AUTH, then reloaded into the LFSR. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be reloaded with RAND_CHALLENGE to prevent a trivial null result.

The 18-bit authentication result AUTH_SIGNATURE is obtained from the final value of CAVE registers R00, R01, R02, R13, R14, and R15. The two most significant bits of AUTH_SIGNATURE are equal to the two least significant bits of R00 XOR R13. The next eight bits of AUTH_SIGNATURE are equal to R01 XOR R14. Finally, the least significant bits of AUTH_SIGNATURE are equal to R02 XOR R15.

If the calling process sets SAVE_REGISTERS to TRUE, the RAND_CHALLENGE, ESN and AUTH_DATA and the contents of the LFSR, offsets and CAVE registers are saved in internal storage. If the calling process sets SAVE_REGISTERS to FALSE, the contents of internal storage are not changed. A means should be provided to indicate whether the internal storage contents are valid.

Exhibit 2-13 Calculation of AUTH SIGNATURE

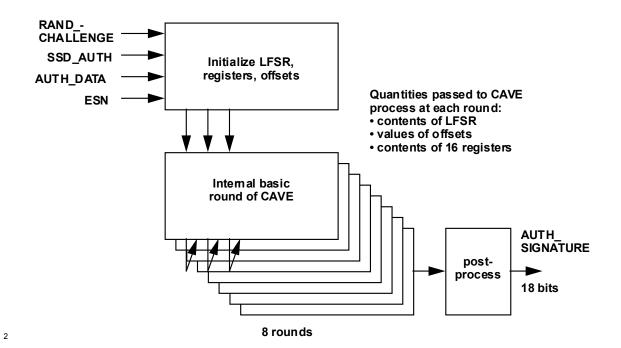


Exhibit 2-14 Code for Calculation of AUTH SIGNATURE

```
2
     /* Auth Signature has the same header as CAVE (see Exhibit 2-4) */
3
4
    unsigned long Auth Signature(const unsigned char RAND_CHALLENGE[4],
5
6
                                       const unsigned char AUTH_DATA[3],
7
                                       const unsigned char *SSD AUTH,
8
                                       const int SAVE REGISTERS)
9
     {
10
        int i,offset_1,offset_2;
        unsigned long AUTH SIGNATURE;
11
12
        for (i = 0; i < 4; i++)
13
14
            LFSR[i] = RAND_CHALLENGE[i] ^ SSD_AUTH[i] ^ SSD_AUTH[i+4];
15
16
17
        if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
18
19
20
            for (i = 0; i < 4; i++)
21
               LFSR[i] = RAND CHALLENGE[i];
22
23
        /* put SSD AUTH into r0-r7 */
24
25
        for (i = 0; i < 8; i++)
26
            Register[i] = SSD AUTH[i];
27
28
        Register[8] = AAV;
29
30
        /* put AUTH DATA into r9-r11 */
31
32
        for (i = 9; i < 12; i++)
33
34
            Register[i] = AUTH DATA[i-9];
35
        /* put ESN into r12-r15 */
36
37
        for (i = 12; i < 16; i++)
38
           Register[i] = ESN[i-12];
39
40
        offset_1 = offset_2 = 128;
41
        CAVE(8, &offset 1, &offset 2);
42
43
        AUTH SIGNATURE =
44
            ( ((unsigned long)(Register[0] ^ Register[13]) << 16) + ((unsigned long)(Register[1] ^ Register[14]) << 8) + ((unsigned long)(Register[2] ^ Register[15]))) & 0x3ffff;
45
46
47
48
```

```
if (SAVE_REGISTERS)
1
2
           /* save LFSR and offsets */
3
           SAVED_OFFSET_1 = offset_1;
           SAVED_OFFSET_2 = offset_2;
           for (\overline{i} = 0; \overline{i} < 4; i++)
8
               SAVED LFSR[i] = LFSR[i];
9
               SAVED RAND[i] = RAND CHALLENGE[i];
10
               if (i < 3)
11
12
                  SAVED DATA[i] = AUTH DATA[i];
13
14
15
        }
16
17
        return(AUTH_SIGNATURE);
18
19
20
```

2.5. Secret Key and Secret Parameter Generation

2

10

11

12

13

16

17

18

19

20

21

23

This section describes four procedures used for generating secret keys and other secret parameters for use in CMEA, Enhanced CMEA (ECMEA) and the voice privacy mask. The generation of distinct secrets for ECMEA-encryption of financial and non-financial messages (e.g. user data) is addressed.

The first procedure uses SSD B and other parameters to generate

- the secret CMEA key for message encryption, and
- the voice privacy mask.

The second procedure uses the secret CMEA key produced in the first procedure to generate the secrets used by ECMEA to encrypt financial messages.

The third procedure uses the secret CMEA key produced in the first procedure to generate the secret non-financial seed key needed to start the fourth procedure.

The fourth procedure uses the secret non-financial seed key produced in the third procedure to generate the secrets used by ECMEA to encrypt non-financial messages.

For backward compatibility with CMEA, the first procedure will always be executed. The secret CMEA key will exist in both the infrastructure and the mobile station.

When ECMEA is implemented, the second, third, and fourth procedures will be executed to produce the secret keys and parameters needed to encrypt both financial and non-financial messages.

2.5.1. CMEA Encryption Key and VPM Generation Procedure

Procedure name:	
Key_VPM_Generation	
Inputs from calling process:	
None.	
Inputs from internal stored data:	
SAVED_LFSR	32 bits
SAVED_OFFSET_1	8 bits
SAVED_OFFSET_2	8 bits
SAVED_RAND	
SAVED_DATA	24 bits
SSD_B	64 bits
AAV	8 bits
Outputs to calling process:	
None.	
Outputs to internal stored data:	
CMEAKEY[0-7]	64 bits
VPM	520 bits

This procedure computes the CMEA key for message encryption and the voice privacy mask. Prior to invoking this procedure, the authentication signature calculation procedure (§2.4) must have been invoked with SAVE_REGISTERS set to TRUE. This procedure must be invoked prior to execution of the encryption procedure (§2.5.2).

The processes for generation of the CMEA key and the voice privacy mask (VPM) will generally be most efficient when concatenated as described in the following sections (§2.5.1.1 and §2.5.1.2). The post-authentication cryptovariables to be used are those from the last authentication signature calculation for which the calling process set SAVE_REGISTERS to true. This should generally be the authentication calculation for the message that establishes the call for which encryption and/or voice privacy is to be invoked. See Exhibit 2-13 and Exhibit 2-14 for graphical detail of the generation process.

2.5.1.1. CMEA key Generation CMEA key generation is depicted in Exhibit 2-16 and Exhibit 2-17. 2 Eight octets of CMEA session key are derived by running CAVE 3 through an 8-round iteration and then two 4-round iterations following an authentication. This is shown in the upper portion of Exhibit 2-16 5 and Exhibit 2-17. The post-authentication initialization and output processing requirements are as follows (for analog phones iterations 4 -14 are omitted: 8 First, the LFSR will be re-initialized to the exclusive-or sum of 9 SAVED LFSR and both halves of SSD B. If the resulting bit 10 pattern fills the LFSR with all zeroes, then the LFSR will be 11 loaded with SAVED RAND. 12 Second, registers R00 through R07 will be initialized with 13 SSD B instead of SSD A. 14 Third, Registers R09, R10, and R11 will be loaded with 15 SAVED DATA. 16 Fourth, Registers R12 through R15 will be loaded with ESN. 17 Fifth, the offset table pointers will begin this process at their 18 (SAVED OFFSET 1 authentication value SAVED OFFSET 2), rather than being reset to a 20 predetermined state. 21 Sixth, the LFSR is loaded before the second and third post-22 authentication iterations with a "roll-over RAND" comprised of 23 the contents of R00, R01, R14, and R15. If the resulting bit 24 pattern fills the LFSR with all zeroes, then the LFSR will be 25 loaded with SAVED RAND. 26 The CMEA key octets drawn from iterations two and three are labeled: 27 k0 = register[4] XOR register[8]; (iteration 2)28 k1 = register[5] XOR register[9]; (iteration 2) 29 k2 = register[6] XOR register[10]; (iteration 2) 30 k3 = register[7] XOR register[11]; (iteration 2) 31 k4 = register[4] XOR register[8]; (iteration 3)32 k5 = register[5] XOR register[9]; (iteration 3) 33 k6 = register[6] XOR register[10]; (iteration 3)34 • k7 = register[7] XOR register[11]; (iteration 3) 35

2.5.1.2. Voice Privacy Mask Generation

VPM generation is a continuation of the CMEA key generation and should be performed at the same time under the same conditions as the CMEA key. CAVE is run for eleven iterations beyond those that produced the CMEA octets. Each iteration consists of four rounds. The CAVE registers R00 through R15 are not reset between iterations, but the LFSR is reloaded between iterations with the "rollover RAND" as described in §2.5.1.1.

Exhibit 2-15 CMEA Key and VPM Generation

```
/* Key VPM Generation has the same header as CAVE (see Exhibit 2-4) */
10
11
    static void roll LFSR(void)
12
13
        int i;
14
15
       LFSR A = Register[0];
16
       LFSR B = Register[1];
17
       LFSR C = Register[14];
18
       LFSR D = Register[15];
19
20
        if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
21
22
           for (i = 0; i < 4; i++)
23
              LFSR[i] = SAVED RAND[i];
24
25
26
27
    void Key VPM Generation(void)
28
29
        int i,j,r ptr,offset 1,offset 2,vpm ptr;
30
31
        /* iteration 1, first pass through CAVE */
32
33
        for (i = 0; i < 4; i++)
34
           LFSR[i] = SAVED LFSR[i] ^ SSD B[i] ^ SSD B[i+4];
35
36
        if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
37
38
           for (i = 0; i < 4; i++)
39
              LFSR[i] = SAVED RAND[i];
40
41
42
        for (i = 0; i < 8; i++)
43
           Register[i] = SSD B[i];
44
45
        Register[8] = AAV;
46
47
        /* put SAVED DATA into r9-r11 */
48
49
        for (i = 9; i < 12; i++)
50
           Register[i] = SAVED DATA[i-9];
51
52
```

```
/* put ESN into r12-r15 */
1
       for (i = 12; i < 16; i++)
           Register[i] = ESN[i-12];
       offset 1 = SAVED OFFSET 1;
6
       offset 2 = SAVED OFFSET 2;
8
       CAVE(8, &offset 1, &offset 2);
9
10
       /* iteration 2, generation of first CMEA key parameters */
11
12
       roll LFSR();
13
       CAVE(4, &offset_1, &offset_2);
14
       for (i = 0; i < 4; i++)
15
           cmeakey[i] = Register[i+4] ^ Register[i+8];
16
17
       /* iteration 3, generation of second CMEA key parameters */
18
19
       roll LFSR();
20
       CAVE(4, &offset 1, &offset 2);
21
22
       for (i = 4; i < 8; i++)
           cmeakey[i] = Register[i] ^ Register[i+4];
23
24
       /* iterations 4-13, generation of VPM */
25
26
27
       vpm ptr = 0;
       for (i = 0; i < 10; i++)
28
29
           roll LFSR();
30
           CAVE(4, &offset_1, &offset_2);
31
           for (r_ptr = 0; r_ptr < 6; r_ptr++)
32
33
              VPM[vpm ptr] = Register[r ptr+2] ^ Register[r ptr+8];
34
              vpm ptr++;
35
36
        }
37
38
       /* iteration 14, generation of last VPM bits */
39
40
       roll LFSR();
41
       CAVE(4, &offset_1, &offset_2);
42
43
       for (j = 0; j < 5; j++)
44
           VPM[vpm ptr] = Register[j+2] ^ Register[j+8];
45
           vpm ptr++;
46
47
48
49
```

Exhibit 2-16 Generation of CMEA Key and VPM

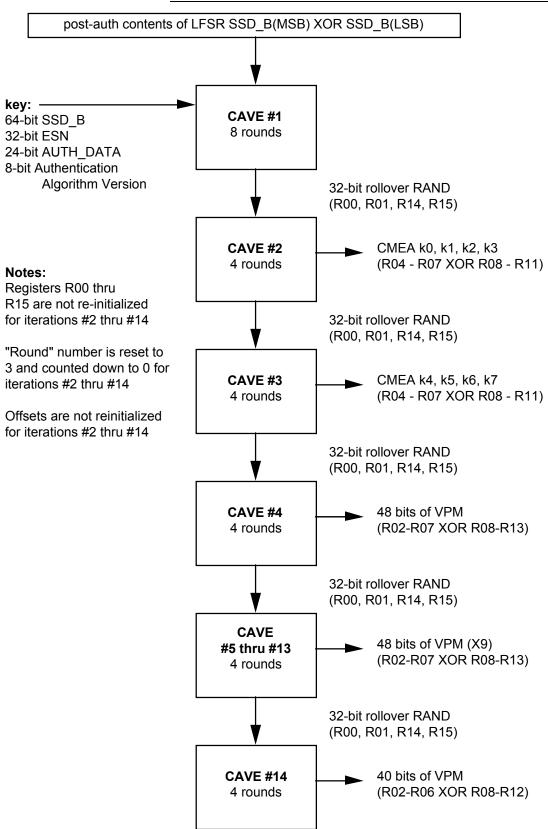
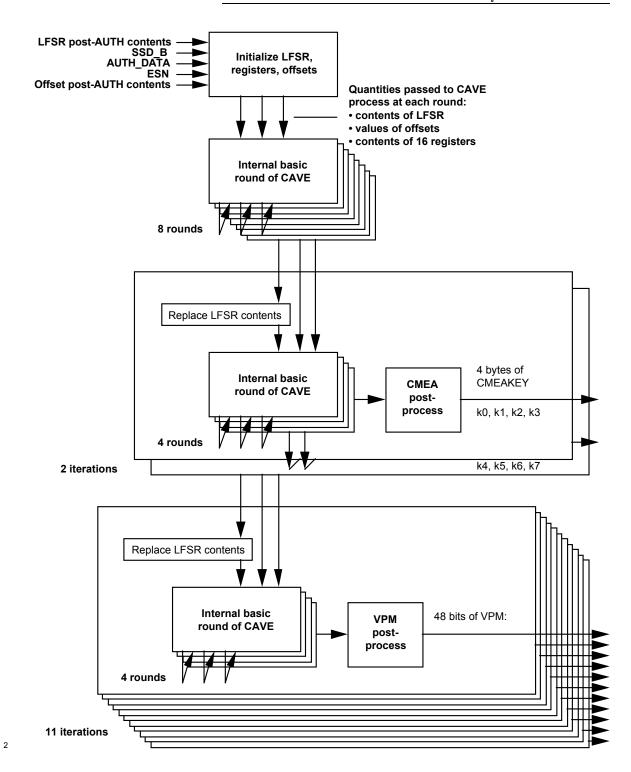


Exhibit 2-17 Detailed Generation of CMEA Key and VPM



38

2.5.2. ECMEA Secrets Generation for Financial Messages **Procedure**

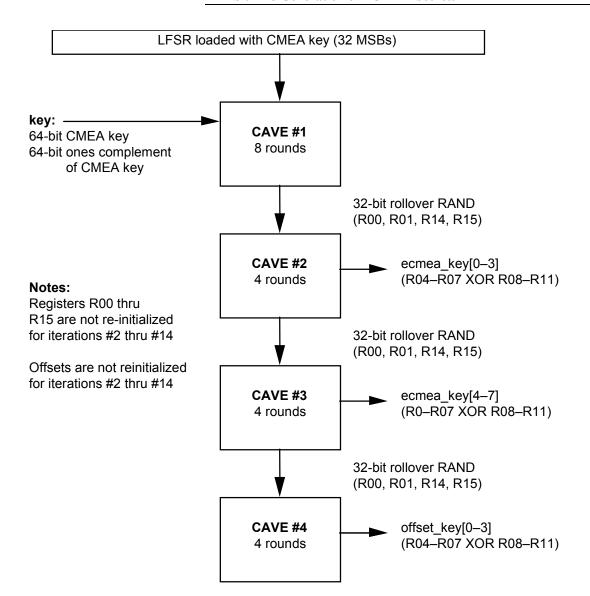
3	Procedure name:	
4	ECMEA_Secret_Generation	
5	Inputs from calling process:	
6	None.	
7	Inputs from internal stored data:	
8	CMEAKEY[0-7] 64 bits	
9	Outputs to calling process:	
10	None.	
11	Outputs to internal stored data:	
12 13	ECMEA_KEY [0-7] 64 bits OFFSET_KEY[0-3] 32 bits	
14 15 16 17	The CMEA Encryption Key and VPM Generation Procedure defined in §2.5.1 is used to generate a CMEA key on a per-call basis. ECMEA for financial messages requires additional secret values to be generated on a per-call basis. This procedure accomplishes this by running the CAVE algorithm initialized by the original CMEA key (64 bits). The generation procedure is depicted in Exhibit 2-18.	
19 20 21 22	 First, the LFSR will be loaded with the 32 MSBs of the CMEA key. If these MSBs are all zero, then a constant, 0x31415926, will be loaded instead. 	
23 24	 Second, registers R00 through R07 will be loaded with the CMEA key. 	
25 26	 Third, registers R08 through R15 will be loaded with the one's-complement of the CMEA key. 	
27	• Fourth, the offset table pointers will be reset to all zeros.	
28	• Fifth, the LFSR is loaded before each of the second through	
29	fourth iterations with a "roll-over RAND" comprised of the	
30	contents of R00, R01, R14, and R15 at the end of the previous	
31	iteration. If the resulting bit pattern fills the LFSR with all	
32 33	zeros, then the LFSR will be loaded with the constant, 0x31415926.	
34 35	The ECMEA key octets drawn from iterations two and three are labelled:	
36	• ecmea_key[0] = register[4] XOR register[8]; (iteration 2)	

• ecmea_key[1] = register[5] XOR register[9]; (iteration 2)

ecmea_key [2] = register[6] XOR register[10]; (iteration 2)

1	• ecmea_key[3] = register[7] XOR register[11]; (iteration 2)
2	ecmea_key[4] = register[4] XOR register[8]; (iteration 3)
3	• ecmea_key[5] = register[5] XOR register[9]; (iteration 3)
4	• ecmea_key[6] = register[6] XOR register[10]; (iteration 3)
5	• ecmea_key[7] = register[7] XOR register[11]; (iteration 3)
6 7 8 9	Note: if, during this process, any of the octets of ECMEA_KEY as defined above are zero, that octet is replaced by the next nonzero octet generated. Additional iterations are performed as necessary to generate eight nonzero octets for ECMEA_KEY.
10	The offset_key octets drawn from iteration 4 are labeled:
11	• offset_key[0] = register[4] XOR register[8]; (iteration 4)
12	• offset_key [1] = register[5] XOR register[9]; (iteration 4)
13	• offset_key [2] = register[6] XOR register[10]; (iteration 4)
14	• offset_key [3] = register[7] XOR register[11]; (iteration 4)

Exhibit 2-18 Generation of ECMEA Secrets



_

Exhibit 2-19 ECMEA Secret Generation

```
/* ECMEA Secret Generation has the same header as ECMEA (see Exhibit 2-
2
3
4
    static void roll LFSR 2(void)
5
6
       LFSR A = Register[0];
7
8
       LFSR_B = Register[1];
9
       LFSR_C = Register[14];
10
       LFSR D = Register[15];
11
       if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
12
13
           LFSR A = 0x31;
14
          LFSR_B = 0x41;
15
          LFSR_C = 0x59;
16
          LFSRD = 0x26;
17
18
    }
19
20
21
    void ECMEA Secret Generation(void)
22
       int i,j,offset 1,offset 2;
23
24
       /* iteration 1, first pass through CAVE */
25
26
       for (i = 0; i < 4; i++)
27
          LFSR[i] = cmeakey[i+4];
28
29
       if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
30
31
           LFSR_A = 0x31;
32
33
           LFSR_B = 0x41;
34
           LFSR_C = 0x59;
           LFSR D = 0x26;
35
36
       for (i = 0; i < 8; i++)
37
          Register[i] = cmeakey[i];
38
39
       for (i = 8; i < 16; i++)
40
           Register[i] = ~cmeakey[i-8];
41
42
       offset_1 = 0x0;
43
       offset_2 = 0x0;
44
45
46
       CAVE(8, &offset 1, &offset 2);
```

```
/* Iterations 2 and 3, generation of ECMEA KEY */
1
       i = 0; j = 4;
3
       while (i < 8)
5
           /* see if new key material needs to be generated */
6
          if(j == 4)
8
              j = 0;
9
              roll LFSR 2();
10
              CAVE(4, &offset 1, &offset 2);
11
12
13
          ecmea_key[i] = Register[j+4] ^ Register[j+8];
14
          j++;
15
16
          /* advance to next octet of ECMEA KEY if not zero; otherwise
17
              generate another value */
18
19
          if (ecmea key[i] != 0)
20
              i++;
21
       }
22
23
       /* iteration 4, generation of ECMEA offset keys */
24
25
26
       roll LFSR 2();
       CAVE(4, &offset_1, &offset_2);
27
       for (i = 0; i < 4; i++)
28
          offset_key[i] = Register[i+4] ^ Register[i+8];
29
30
31
```

2.5.3. Non-Financial Seed Key Generation Procedure

2	Procedure name:
3	Non-Financial_Seed_Key_Generation
4	Inputs from calling process:
5	None.
6	Inputs from internal stored data:
7	CMEAKEY[0-7] 64 bits
8	Outputs to calling process:
9	None.
10	Outputs to internal stored data:
11	SEED_NF_KEY[0-4] 40 bits
12	The CMEA Encryption Key and VPM Generation Procedure defined in
13	§2.5.1 is used to generate a CMEA key on a per-call basis. A non-
14	financial seed key is required before generating the ECMEA secrets for
15	non-financial messages. This procedure accomplishes this by running
16	the CAVE algorithm initialized by the original CMEA key (64 bits).
17	The generation procedure is depicted in Exhibit 2-20.
18	 First, the LFSR will be loaded with the 32 LSBs of the CMEA
19	key. If these MSBs are all zero, then a constant, 0x31415926,
20	will be loaded instead.

21

22

23

24

25

26

27

29

30

31

32

33

34

35

36

- Second, registers R00 through R07 will be loaded with the one's-complement of the CMEA key.
- Third, registers R08 through R15 will be loaded with the CMEA key.
- Fourth, the offset table pointers will be reset to all zeros.
- Fifth, the LFSR is loaded before the second iteration with a "roll-over RAND" comprised of the contents of R00, R01, R14, and R15 at the end of the previous iteration. If the resulting bit pattern fills the LFSR with all zeros, then the LFSR will be loaded with the constant, 0x31415926.

The non-financial seed key octets drawn from iteration two are labeled:

- seed nf key[0] = register[2] XOR register[8]; (iteration 2)
- seed_nf_key[1] = register[3] XOR register[9]; (iteration 2)
- seed_nf_key[2] = register[4] XOR register[10]; (iteration 2)
- seed nf key[3] = register[5] XOR register[11]; (iteration 2)
- seed_nf_key [4] = register[6] XOR register[12]; (iteration 2)

Exhibit 2-20 Generation of Non-Financial Seed Key

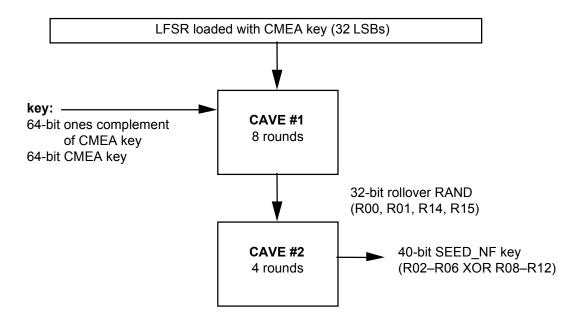


Exhibit 2-21 Non-Financial Seed Key Generation

```
/* Non Financial Seed Key Generation has the same header as ECMEA (see
2
    Exhibit 2-30) */
3
4
    void Non Financial Seed Key Generation(void)
5
6
       int i,offset 1,offset 2;
7
8
9
       /* iteration 1, first pass through CAVE */
10
       for (i = 0; i < 4; i++)
11
          LFSR[i] = cmeakey[i];
12
13
       if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
14
15
          LFSR A = 0x31;
16
          LFSR^{-}B = 0x41;
17
          LFSR_C = 0x59;
18
          LFSR D = 0x26;
19
20
21
       for (i = 0; i < 8; i++)
          Register[i] = ~cmeakey[i];
22
23
       for (i = 8; i < 16; i++)
24
          Register[i] = cmeakey[i-8];
25
26
       offset_1 = 0x0;
27
       offset 2 = 0x0;
28
29
       CAVE(8, &offset 1, &offset 2);
30
31
       /* iteration 2, generation of seed nf key */
32
33
34
       roll_LFSR_2(); /* defined in Exhibit 2-19 */
       CAVE(4, &offset_1, &offset_2);
35
       for (i = 0; i < 5; i++)
36
          seed nf key[i] = Register[i+2] ^ Register[i+8];
37
    }
38
```

2.5.4. ECMEA Secrets Generation for Non-Financial Messages Procedure

3	Procedure name:	
4	Non-Financial_Secret_Generation	
5	Inputs from calling process:	
-	mp wio 110111 0 mm 18 p 1000 0 0 0 1	
3	None.	
7	Inputs from internal stored data:	
•	The way in the state of the sta	
3	SEED_NF_KEY[0-4] 40 bits	
	Outputs to calling process:	
•	Sulputs to curing process.	
)	None.	
1	Outputs to internal stored data:	
	Sulputo to internal stored data.	
2	ECMEA_NF_KEY[0-7] 64 bits	
3	OFFSET_NF_KEY[0-3] 32 bits	
4	The Non-Financial Seed Key Generation Procedure defined in §2.5.3 is	

The Non-Financial Seed Key Generation Procedure defined in §2.5.3 is used to generate a seed key on a per-call basis. ECMEA for non-financial messages requires additional secret values to be generated on a per-call basis. This procedure accomplishes this by running the CAVE algorithm initialized by the original seed key (40 bits). The generation procedure is depicted in Exhibit 2-22.

- First, the LFSR will be loaded with the 32 MSBs of the SEED_NF key. If these MSBs are all zero, then a constant, 0x31415926, will be loaded instead.
- Second, registers R00 through R04 will be loaded with the 40-bit SEED NF key.
- Third, registers R05 through R07 will be loaded with zeros.
- Fourth, registers R08 through R12 will be loaded with the one's-complement of the 40-bit SEED NF key.
- Fifth, registers R13 through R15 will be loaded with zeros.
- Sixth, the offset table pointers will be reset to all zeros.
- Seventh, the LFSR is loaded before each of the second through seventh iterations with a "roll-over RAND" comprised of the contents of R00, R01, R14, and R15 at the end of the previous iteration. If the resulting bit pattern fills the LFSR with all zeros, then the LFSR will be loaded with the constant, 0x31415926.

1 2	The ECMEA_NF key octets drawn from iterations two and three are labeled:
3	• ecmea_nf_key[0] = register[4] XOR register[8]; (iteration 2)
4	• ecmea_nf_key[1] = register[5] XOR register[9]; (iteration 2)
5	• ecmea_nf_key[2] = register[6] XOR register[10]; (iteration 2)
6	• ecmea_nf_key[3] = register[7] XOR register[11]; (iteration 2)
7	• ecmea_nf_key[4] = register[4] XOR register[8]; (iteration 3)
8	• ecmea_nf_key[5] = register[5] XOR register[9]; (iteration 3)
9	• ecmea_nf_key[6] = register[6] XOR register[10]; (iteration 3)
10	• ecmea_nf_key[7] = register[7] XOR register[11]; (iteration 3)
11	Note: if, during this process, any of the octets of ECMEA_NF_KEY as
12	defined above are zero, that octet is replaced by the next nonzero octet
13 14	generated. Additional iterations are performed as necessary to generate eight nonzero octets for ECMEA NF KEY.
	~
15	The offset_key octets drawn from iteration 4 are labeled:
16	• offset_nf_key[0] = register[4] XOR register[8]; (iteration 4)
17	 offset_nf_key[1] = register[5] XOR register[9]; (iteration 4)
18	• offset_nf_key[2] = register[6] XOR register[10]; (iteration 4)
19	• offset_nf_key3] = register[7] XOR register[11]; (iteration 4)

3

Exhibit 2-22 Generation of Non-Financial Secrets

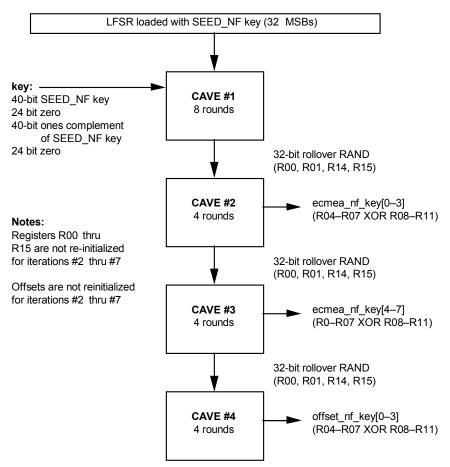


Exhibit 2-23 Non-Financial Secret Generation

```
/* Non Financial Secret Generation has the same header as ECMEA (see
4
    Exhibit 2-30) */
5
6
    void Non Financial Secret Generation(void)
7
8
       int i,j,offset 1,offset 2;
9
10
       /* iteration 1, first pass through CAVE */
11
12
       for (i = 0; i < 4; i++)
13
          LFSR[i] = seed nf key[i+1];
14
15
       if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
16
17
          LFSR A = 0x31;
18
          LFSR_B = 0x41;
19
          LFSRC = 0x59;
20
          LFSR^{-}D = 0x26;
21
22
```

```
for (i = 0; i < 5; i++)
1
           Register[i] = seed_nf_key[i];
       for (i = 5; i < 8; i++)
           Register[i] = 0;
6
       for (i = 8; i < 13; i++)
           Register[i] = ~seed nf key[i-8];
8
9
       for (i = 13; i < 16; i++)
10
           Register[i] = 0;
11
12
       offset 1 = 0x0;
13
       offset 2 = 0x0;
14
15
       CAVE(8, &offset 1, &offset 2);
16
17
       /* Iterations 2 and 3, generation of ECMEA NF KEY */
18
19
       i = 0; i = 4;
20
       while (i < 8)
21
22
           /* see if new key material needs to be generated */
23
           if(j == 4)
24
25
              j = 0;
26
              roll LFSR_2();
27
              CAVE(4, &offset_1, &offset_2);
28
29
30
           ecmea nf key[i] = Register[j+4] ^ Register[j+8];
31
           j++;
32
33
           /* advance to next octet of ECMEA NF KEY if not zero; otherwise
34
              generate another value */
35
36
           if (ecmea nf key[i] != 0)
37
              i++;
38
       }
39
40
       /* iteration 4, generation of ECMEA offset nf key */
41
42
43
       roll LFSR 2(); /* defined in Exhibit 2-19 */
       CAVE(4, &offset 1, &offset 2);
44
       for (i = 0; i < 4; i++)
45
           offset nf key[i] = Register[i+4] ^ Register[i+8];
46
47
48
49
```

Message Encryption/Decryption Procedures 2.6.

2.6.1. CMEA Encryption/Decryption Procedure

3	Procedure name:	
4	Encrypt	
5	Inputs from calling process:	
6	$msg_buf[n]$ $n*8 bits, n > 1$	
7	Inputs from internal stored data:	
8	CMEAKEY[0-7] 64 bits	
9	Outputs to calling process:	
10	msg_buf[n] n*8 bits	
11	Outputs to internal stored data:	
12	None.	
13 14 15	This algorithm encrypts and decrypts messages that are of length $n*8$ bits, where $n>1$. Decryption is performed in the same manner as encryption.	
16 17 18 19	The message is first stored in an n-octet buffer called msg_buf[], such that each octet is assigned to one "msg_buf[]" value. msg_buf[] will be encrypted by means of three operations before it is ready for transmission.	
20 21 22	This process uses the CMEA eight-octet session key to produce enciphered messages via a unique CMEA algorithm. The process of CMEA key generation is described in §2.5.1.	
23	The function tbox() is frequently used. This is defined as:	
24	tbox(z) = C(((C(((C(((C(((C(((C(((C(((C((C((C((C(
25	where "+" denotes modulo 256 addition,	
26	"XOR" is the XOR function,	
27	"z" is the function argument,	
28	k0,,k7 are defined above,	
29 30	and $C(\)$ is the outcome of a CAVE 8-bit table look-up, (Exhibit 2-5)	

Exhibit 2-24 shows ANSI C code for an algorithmic procedure for tbox().

Exhibit 2-24 tbox

2

3

25

26

27

28

29

```
/* tbox has the same header as CAVE (see Exhibit 2-4) */
5
    static unsigned char tbox(const unsigned char z)
6
7
       int k index,i;
8
       unsigned char result;
9
10
       k index = 0;
11
       result = z;
12
13
       for (i = 0; i < 4; i++)
14
15
           result ^= cmeakey[k_index];
16
           result += cmeakey[k index+1];
17
           result = z + CaveTable[result];
18
19
           k index += 2;
20
21
       return(result);
22
23
24
```

The CMEA algorithm is the message encryption process used for both the encryption and decryption of a message. Each message to which the CMEA algorithm is applied must be a multiple of 8 bits in length. The CMEA algorithm may be divided into three distinct manipulations. See Exhibit 2-25.

Exhibit 2-25 CMEA Algorithm

```
/* CMEA has the same header as CAVE (see Exhibit 2-4) */
2
3
4
    void CMEA(unsigned char *msg buf, const int octet count)
5
       int msg_index,half;
6
       unsigned char k,z;
7
8
9
       /* first manipulation (inverse of third) */
10
11
       z = 0;
       for (msg index = 0; msg index < octet count; msg index++)</pre>
12
13
           k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
14
           msg buf[msg index] += k;
15
           z += msg buf[msg index];
16
17
18
        /* second manipulation (self-inverse) */
19
20
21
       half = octet_count/2;
       for (msg index = 0; msg index < half; msg index++)</pre>
22
23
           msg_buf[msg_index] ^=
24
              msg_buf[octet_count - 1 - msg_index] | 0x01;
25
26
27
        /* third manipulation (inverse of first) */
28
29
30
       z = 0;
31
       for (msg index = 0; msg index < octet count; msg index++)</pre>
32
           k = tbox((unsigned char)(z ^ (msg_index & 0xff)));
33
34
           z += msg_buf[msg_index];
           msg_buf[msg_index] -= k;
35
36
37
38
```

2.6.2. ECMEA Encryption/Decryption Procedure

2	Procedure name:		
3	ECMEA		
•	LCMEA		
4	Inputs from calling process:		
5	msg_buf[n]	n*8 bits, n > 1	
6	Sync[0-1]	16 bits	
7	Decrypt	1 bit	
8	Data_type	1 bit	
9	Inputs from internal stored data:	Inputs from internal stored data:	
10	ECMEA_KEY[0-7]	64 bits	
11	offset_key[0-3]	32 bits	
12	Outputs to calling process:		
12			
13	msg_buf[n]	n*8 bits	
14	Outputs to internal stored data:		
15	None.		
16	This algorithm encrypts and decr	rypts messages that are of length n*8	
17	bits, where $n > 1$.		
40	The massage is first stored in a	a n actat buffer called mag buffl	
18 19		n n-octet buffer called msg_buf[],	
20		such that each octet is assigned to one "msg_buf[]" value. The input variable sync should have a unique value for each message that is	
21		encrypted. The same value of sync is used again for decryption.	
22		eight-octet session key to produce	
23		enciphered messages via an enhanced CMEA algorithm. The process	
24	of ECMEA key generation is desc	Hoeu III 82.3.2.	
25	The decrypt variable shall be	set to 0 for encryption and to 1 for	
26	The decrypt variable shall be set to 0 for encryption, and to 1 for decryption.		
	•		
27	The data type variable shall be set to 0 for financial messages, and		
28	to 1 for non-financial messages.		
29	ECMEA encryption of financial messages uses ECMEA key and		
30	offset_key.		
31	ECMEA encryption of non-financial messages uses ECMEA_NF key		
32	and offset_nf_key.		

```
The function etbox() is frequently used. This is defined as:
1
     etbox(z,k) = I(I(I(I(I(I(I(I(I(I(I(I(I(z+k0)XOR k1)+k2)XOR k3)+k4)XOR k5)+k6)XOR k7)-k6)XOR k5)
2
                                      k4)XOR k3)-k2)XOR k1)-k0
3
                                       where "+" denotes modulo 256 addition,
                                              "-" denotes modulo 256 subtraction,
                                              "XOR" is the XOR function,
                                              "z" is the function argument,
                                              k0,...,k7 are the eight octets of ECMEA key,
8
                                       and I() is the outcome of the ibox 8-bit table look-up, (Exhibit
9
                                       2-2).
10
                                   Exhibit 2-26 shows ANSI C code for an algorithmic procedure for
11
12
                                   Exhibit 2-26 Enhanced tbox
13
     /* enhanced tbox has the same header as ECMEA (see Exhibit 2-30) */
14
15
     unsigned char etbox(const unsigned char z,
16
     const unsigned char *ecmea key)
17
18
19
        unsigned char t;
20
21
         t = ibox[(z + ecmea_key[0]) & 0xff];
t = ibox[t ^ ecmea_key[1]];
22
23
         t = ibox[(t + ecmea key[2]) & 0xff];
24
         t = ibox[t ^ ecmea_key[3]];
25
         t = ibox[(t + ecmea_key[4]) & 0xff];
26
         t = ibox[t ^ ecmea \overline{key}[5]];
27
         t = ibox[(t + ecmea key[6]) & 0xff];
28
         t = ibox[t ^ ecmea_key[7]];
29
         t = ibox[(t - ecmea key[6]) & 0xff];
30
         t = ibox[t ^ ecmea_key[5]];
31
         t = ibox[(t - ecmea key[4]) & 0xff];
32
         t = ibox[t ^ ecmea key[3]];
33
         t = ibox[(t - ecmea key[2]) & 0xff];
34
        t = ibox[t ^ ecmea_key[1]];
35
         t = (t - ecmea key[0]) & 0xff;
36
37
38
         return t;
39
40
```

21

22

23

Enhanced CMEA is based on the basic CMEA construct for ease of implementation. It uses a modified CMEA which is passed keying information. The ECMEA encryption algorithm also uses a transformation and its inverse which are called before and after the CMEA block.

For each message encrypted or decrypted with ECMEA, offsets are calculated and then used to permute the tbox values used in CMEA and the transformations. ECMEA uses two offsets which are calculated as follows:

```
offset12 = ((offset_key[1.0]+1)*(CS[1,0] + 1) mod 65537)

XOR offset_key[3,2]

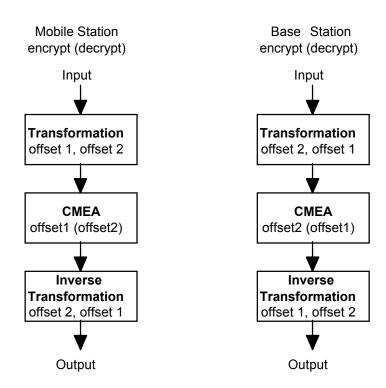
offset1 = (offset12 >> 8) mod 256
```

offset2 = offset1 XOR MAX(offset12 mod 256, 1)

where XOR stands for logical bitwise exclusive or, offset_key[i,j] means octets i and j of offset_key concatenated to form a 16-bit quantity with the second octet as the least significant, and CS denotes the 16 bits of cryptosynchronizing information for the message.

CMEA uses one offset while the transformation and its inverse use two offsets. The transformations are non-self-inverting and so the entire algorithm is non-self-inverting. For the inverse ECMEA algorithm, the order of passing offsets to the transformations is reversed. ECMEA is configured as shown in Exhibit 2-27.

Exhibit 2-27 ECMEA Structure



The mobile station and the base station implement the same basic algorithm with the only change being the offsets that are used in the transformation, in CMEA and in the inverse transformation. For example, offsets 1 and 2 (in that order) are used in the first transformation in the mobile station while the same offsets in the reverse order are passed to the first transformation in the base station. The inverse transformation always uses the offsets in the reverse order. The transformation and its inverse are given in Exhibit 2-28.

Exhibit 2-28 ECMEA Transformation and its Inverse

```
/* transform and inv transform have the same header as ECMEA (see
2
    Exhibit 2-30) */
3
4
    void transform(unsigned char *msg buf, const int octet count,
5
                    unsigned char offseta, const unsigned char offsetb,
6
7
                    const unsigned char *key)
8
9
       unsigned char k, z;
10
       int msg_index;
11
       for (msq index = 0; msq index < octet count; msq index++)
12
13
           /* offseta rotation and involutary lookup of present octet */
14
15
          if (msg_index > 0)
16
             offseta = (offseta >> 1) | (offseta << 7);
17
          msg_buf[msg_index] = offsetb
18
              etbox((unsigned char) (msg buf[msg index] ^ offseta), key);
19
20
21
          /* bit-trade between present octet and the one below */
22
          if (msg index > 0)
23
24
             k = msg_buf[msg_index - 1] ^ msg_buf[msg_index];
25
             k &= etbox((unsigned char)(k ^ offseta), key);
26
             msg_buf[msg_index - 1] ^= k;
27
             msg_buf[msg_index] ^= k;
28
29
30
           /* random octet permutation */
31
          /* exchange previous octet with a random one below it */
32
33
          if (msg index > 1)
34
35
             k = etbox((unsigned char)(msg buf[msg index] ^ offseta),
36
                 key);
37
             k = ((msg\_index) * k) >> 8;
38
              z = msg buf[k];
39
             msg buf[k] = msg buf[msg index - 1];
40
             msg buf [msg index - 1] = z;
41
42
43
44
       /* final octet permutation */
45
       /* exchange last octet with a random one below it */
46
47
       k = etbox((unsigned char)(0x37 ^ offseta), key);
48
       k = ((msg index) * k) >> 8;
49
       z = msg_buf[k];
50
       msg_buf[k] = msg_buf[msg_index - 1];
51
       msg buf [msg index - 1] = z;
52
53
```

```
/* final involution and XORing */
1
       k = etbox(msg buf[0], key);
       for (msg index = 1; msg index < octet count; msg index++)</pre>
           msq buf[msq index] = etbox(msq buf[msq index], key);
6
           k = msq buf[msq index];
8
9
       msq buf[0] = k;
10
       for (msg index = 1; msg index < octet count; msg index++)</pre>
11
           msg buf[msg index] ^= k ;
12
13
14
15
    /* Inverse Transformation */
16
17
    void inv transform (unsigned char *msg buf, const int octet count,
18
                         unsigned char offseta, const unsigned char offsetb,
19
                         const unsigned char *key)
20
21
       unsigned char k, z;
22
       int msg index;
23
24
       /* initial offseta rotation */
25
26
       k = (\text{octet count} - 1) \& 0x07;
27
       offseta = (offseta >> k) | (offseta << (8 - k));
28
29
       /* inverse of final involution and XORing */
30
31
       for (msg_index = 1; msg_index < octet_count; msg_index++)</pre>
32
           msg buf[msg index] ^= msg buf[0];
33
34
       for (msg index = 1; msg index < octet count; msg index++)</pre>
35
36
           msg buf[0] ^= msg buf[msg index];
37
           msg buf[msg index] = etbox(msg buf[msg index], key);
38
39
       msg buf[0] = etbox(msg buf[0], key);
40
41
        /* initial octet permutation */
42
       /* exchange last octet with a random one below it */
43
44
       k = etbox((unsigned char)(0x37 ^ offseta), key);
45
       k = ((octet count) * k) >> 8;
46
       z = msq buf[k];
47
       msg_buf[k] = msg_buf[octet count - 1];
48
       msq buf[octet count - 1] = z;
49
50
```

```
for (msg index = octet count - 1; msg index >= 0; msg index--)
1
2
           /* random octet permutation */
3
           /* exchange previous octet with a random one below it */
           if (msg index > 1)
              k = etbox((unsigned char)(msg buf[msg index] ^ offseta),
8
9
              k = ((msg index) * k) >> 8;
10
              z = msq buf[k];
11
              msg buf[k] = msg buf[msg index - 1];
12
              msg buf[msg index - 1] = z;
13
14
15
           /* bit-trade between present octet and the one below */
16
17
           if (msg index > 0)
18
19
              k = msg_buf[msg_index - 1] ^ msg_buf[msg_index];
20
              k &= etbox((unsigned char)(k ^ offseta), key);
21
              msg_buf[msg_index - 1] ^= k;
msg_buf[msg_index] ^= k;
22
23
24
25
           /* involutary lookup of present octet and offset rotation */
26
27
           msg_buf[msg_index] = offseta ^
28
              etbox((unsigned char)(msg_buf[msg_index] ^ offsetb), key);
29
           offseta = (offseta << 1) | (offseta >> 7);
30
       }
31
32
33
```

2

3

4

5

6

8

9

10

11

12

13

14

15

Exhibit 2-30 gives the ECMEA algorithm for the mobile station. Each message to which ECMEA is applied must be a multiple of 8 bits in length.

The base station algorithm is the same as the mobile station algorithm except for the two calls to the transformations and the offset used for CMEA. C code for the base station procedure is identical to that in Exhibit 2-30, except the first transformation call is changed to

the offsets used for CMEA are reversed (i.e., the decryption and encryption offsets are the opposite of those used by the mobile station) and the final inverse transformation call is changed to

Exhibit 2-29 ECMEA Algorithm Header

```
void ECMEA Secret Generation(void);
16
17
    void Non Financial Seed Key Generation (void);
18
19
    void Non Financial Secret Generation(void);
20
21
    void ECMEA(unsigned char *msg buf,
22
                const int octet count,
23
                const unsigned char sync[2],
24
                const unsigned int decrypt,
25
26
                const unsigned int data type);
27
28
    #ifndef ECMEA SOURCE FILE
29
    extern
                        ecmea key[8];
30
    unsigned char
31
    extern
    unsigned char
                        ecmea nf key[8];
32
33
    extern
    unsigned char
                        offset key[4];
34
35
    extern
                       offset nf key[4];
    unsigned char
36
37
    extern
38
    unsigned char
                       seed nf key[5];
39
    #endif
40
```

Exhibit 2-30 ECMEA Encryption/Decryption Algorithm for the Mobile Station

```
#define ECMEA SOURCE FILE
3
    #include "cavei.h" /* see Exhibit 2-3 */
4
    #include "ecmea.h" /* see Exhibit 2-29 */
5
6
    #define MOBILE 1 /* set to 0 for base station algorithm */
7
8
    void ECMEA(unsigned char *msg buf, const int octet count,
9
                const unsigned char sync[2],
10
11
                const unsigned int decrypt,
                const unsigned int data type)
12
    {
13
       unsigned char k, z, offset1, offset2, offsetc;
14
       unsigned long x1, x2, s;
15
       int msg index;
16
       unsigned char *key, *offset;
17
18
       /* select key and offset key */
19
       if (data type)
20
21
22
          key = ecmea nf key;
          offset = offset nf key;
23
       }
24
       else
25
26
       {
          key = ecmea key;
27
          offset = offset key;
28
29
30
       /* calculate offsets */
31
       /* offset12 =
32
           ((offset[1,0]+1)*(CS+1) mod 65537) offset[3,2] mod 65536 */
33
       x1 = ((unsigned long)offset[1] << 8) + (unsigned long)offset[0];</pre>
34
       x2 = ((unsigned long)offset[3] << 8) + (unsigned long)offset[2];</pre>
35
       s = ((unsigned long)sync[1] << 8) + (unsigned long)sync[0];
36
       /* x1 = (((x1 + 1) * (s + 1)) % 65537) ^ x2; in two steps to
37
          prevent overflow */
38
       x1 = (x1 * (s + 1)) % 65537;
39
       x1 = ((x1 + s + 1) % 65537)^{-} x2;
40
       offset1 = (unsigned char) (x1 >> 8);
41
       offset2 = (unsigned char) (offset1 ^ x1);
42
       if (offset2 == offset1)
43
          offset2 ^= 1;
44
45
    #if MOBILE
46
47
       if (decrypt)
48
49
          offsetc = offset2;
50
       else
          offsetc = offset1;
51
```

2

```
#else
1
2
       if (decrypt)
3
          offsetc = offset1;
       else
5
          offsetc = offset2;
6
7
    #endif
8
9
       /* initial transformation */
10
    #if MOBILE
11
       transform(msg buf, octet count, offset1, offset2, key);
12
13
       transform(msg buf, octet count, offset2, offset1, key);
14
    #endif
15
16
       /* CMEA */
17
        /* first manipulation (inverse of third) */
18
       z = 0;
19
       for (msq index = 0; msq index < octet count; msq index++)
20
21
           k = etbox((unsigned char)(z ^ offsetc), key);
22
           msq buf[msq index] += k;
23
           z = msq buf[msq index];
24
25
26
        /* second manipulation (self-inverse) */
27
       for (msg_index = 0; msg_index < octet_count - 1; msg_index += 2)</pre>
28
           msg buf[msg index] ^= msg buf[msg index + 1];
29
30
        /* third manipulation (inverse of first) */
31
       z = 0;
32
       for (msg index = 0; msg index < octet count; msg index++)</pre>
33
34
           k = etbox((unsigned char)(z ^ offsetc), key);
35
           z = msg buf[msg_index];
36
           msg buf[msg index] -= k;
37
38
39
       /* final inverse transformation */
40
    #if MOBILE
41
       inv transform(msg buf, octet count, offset2, offset1, key);
42
43
       inv transform(msg buf, octet count, offset1, offset2, key);
44
    #endif
45
46
47
```

2.7. Wireless Residential Extension Procedures

1

2

6

8

This section describes detailed cryptographic procedures for wireless mobile telecommunications systems offering auxiliary services. These procedures are used to perform the security services of Authorization and Call Routing Equipment (ACRE), Personal Base (PB) and Mobile Station (MS) authentication. The ANSI C header file for Wireless Residential Extension Procedures is given in

Exhibit 2-31 WRE Header

```
9
    void WIKEY Generation(const unsigned char MANUFACT KEY[16],
                           const unsigned char PBID[4]);
10
11
    void WIKEY Update(const unsigned char RANDWIKEY[7],
12
                       const unsigned char PBID[4]);
13
14
    unsigned long WI Auth Signature (const unsigned char RAND CHALLENGE [4],
15
                                     const unsigned char PBID[4],
16
17
                                     const unsigned char
    ACRE PHONE NUMBER[3]);
18
19
20
    unsigned long WRE Auth Signature(const unsigned char RAND WRE[3],
21
                                       const unsigned char PBID[4],
22
                                       const unsigned char ESN[4]);
23
24
    #ifndef WRE SOURCE FILE
25
    extern
   unsigned char
26
                      WIKEY[8];
27
   extern
   unsigned char WIKEY_NEW[8];
    extern
    unsigned char
                      WRE KEY[8];
    #endif
31
```

2.7.1. WIKEY Generation

2	Procedure name:	
3	WIKEY_Generation	
4	Inputs from calling process:	
5	MANUFACT_KEY	122 bits
6	PBID	30 bits
7	Inputs from internal stored data:	
8	AAV	8 bits
9	Outputs to calling process:	
10	None.	
11	Outputs to internal stored data:	
12	WIKEY	64 bits
	This procedure is used to coloulate	the WIVEV value concreted during
13 14		e the WIKEY value generated during s WIKEY value is stored in semi-
15	permanent memory of the PB.	S WIKE I VALUE IS STOLED III SCHILL
16	The initial loading of CAVE for	calculation of WIKEY is given in
17	Exhibit 2-32.	calculation of WHYET is given in
10	MANUFACT KEV is a 122-b	ait value that is chosen by the
18 19	MANUFACT_KEY is a 122-bit value that is chosen by the manufacturer. This value is the same for all of the manufacturer's PBs.	
20	PB manufactures must provide this number to each ACRE manufacture	
21		the correct WIKEY values. The 32
22	MSBs of MANUFACT_KEY must not be all zeroes. There must be at	
23	least 40 zeroes and 40 ones in MANUFACT_KEY.	

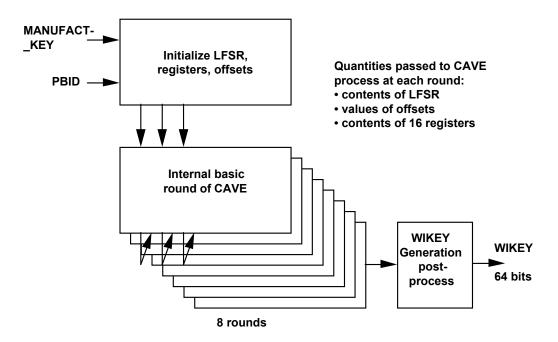
Exhibit 2-32 CAVE Initial Loading for WIKEY Generation

CAVE Item	Source Identifier	Size (Bits)
LFSR	bits 121-90 (32 MSBs) of MANUFACT_KEY	32
Reg [0-7]	bits 89-26 of MANUFACT_KEY	64
Reg [8]	AAV	8
Reg [9-11]	bits 25-2 of MANUFACT_KEY	24
Reg [12] 2 MSBs	bits 1-0 (2 LSBs) of MANUFACT_KEY	2
Reg [12] 6 LSBs	6 MSBs of PBID	6
Reg [13-15]	24 LSBs of PBID	24

CAVE is run for eight rounds. The 64-bit result is WIKEY. Exhibit 2-33 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-34.

The 64-bit WIKEY result is obtained from the final value of CAVE registers R00 through R15. The first 8 CAVE registers are XORed with the last 8 CAVE registers to produce the value for WIKEY.

Exhibit 2-33 Generation of WIKEY



10

Exhibit 2-34 Code for WIKEY Generation

```
#define WRE SOURCE FILE
2
    #include "cavei.h" /* see Exhibit 2-3 */
3
                       /* see Exhibit 2-31 */
    #include "wre.h"
    unsigned char
                      WIKEY[8];
6
    unsigned char
                       WIKEY_NEW[8];
7
8
    unsigned char
                       WRE KEY[8];
10
    /* Note that MANUFACT KEY is left justified and PBID is right justified.
       This means that the 6 LSBs of MANUFACT KEY and the 2 MSBs of PBID
11
       must be set to 0 by the calling routine. */
12
13
    void WIKEY Generation(const unsigned char MANUFACT KEY[16],
14
                           const unsigned char PBID[4])
15
16
       int i,offset 1,offset 2;
17
18
       for (i = 0; i < 4; i++)
19
20
          LFSR[i] = MANUFACT KEY[i];
21
       for (i = 0; i < 8; i++)
          Register[i] = MANUFACT KEY[i+4];
22
       Register[8] = AAV;
23
       for (i = 0; i < 4; i++)
24
          Register[i+9] = MANUFACT_KEY[i+12];
25
26
       Register[12] = Register[12] | PBID[0];
       for (i = 0; i < 3; i++)
27
          Register[i+13] = PBID[i+1];
28
       offset_1 = offset_2 = 128;
29
       CAVE(8, &offset_1, &offset_2);
30
31
       for (i = 0; i < 8; i++)
          WIKEY[i] = Register[i] ^ Register[i+8];
32
33
34
```

2.7.2. WIKEY Update Procedure

Procedure name: WIKEY_Update Inputs from calling process: **RANDWIKEY** 56 bits **PBID** 30 bits Inputs from internal stored data: 64 bits WIKEY 8 bits AAV Outputs to calling process: None. Outputs to internal stored data: WIKEY NEW 64 bits

This procedure is used to calculate a new WIKEY value.

The initial loading of CAVE for calculation of WIKEY_NEW is given in Exhibit 2-35.

Exhibit 2-35 CAVE Initial Loading for WIKEY Update

CAVE Item	Source Identifier	Size (Bits)
LFSR	32 LSB of RANDWIKEY	32
Reg [0-7]	WIKEY	64
Reg [8]	AAV	8
Reg [9-11]	24 MSB of RANDWIKEY	24
Reg [12] 2 MSBs	00	2
Reg [12] 6 LSBs	6 MSBs of PBID	6
Reg [13-15]	24 LSBs of PBID	24

CAVE is run for eight rounds. The 64-bit result is WIKEY_NEW. Exhibit 2-36 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-37.

The LFSR will initially be loaded with the 32 LSBs of RANDWIKEY. This value will be XOR'd with the 32 most significant bits of WIKEY XOR'd with the 32 least significant bits of WIKEY, then reloaded into the LFSR. If the resulting bit pattern fills the LFSR with all zeroes, then

18

10

11

12

13

14

15

16

17

19 20

21

22 23

5

7

8

the LFSR will be reloaded with the 32 LSBs of RANDWIKEY to prevent a trivial null result.

The 64-bit WIKEY NEW result is obtained from the final value of CAVE registers R00 through R15. The first 8 CAVE registers are XORed with the last 8 CAVE registers to produce the value for WIKEY NEW.

Exhibit 2-36 Generation of WIKEY NEW

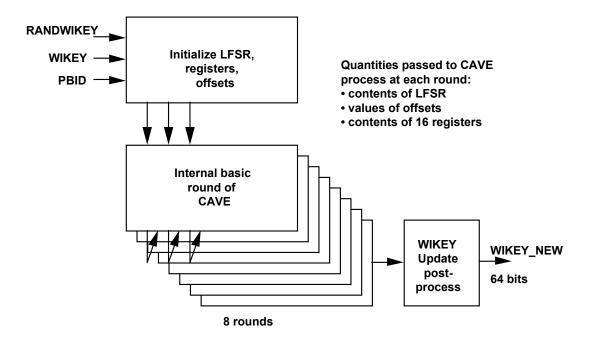


Exhibit 2-37 Code for WIKEY NEW Generation

```
/* WIKEY Update has the same header as WIKEY Generation (see Exhibit 2-
3
    34) */
4
6
    /* Note that PBID is right justified. This means that the 2 MSBs of PBID
       must be set to 0 by the calling routine. */
7
8
    void WIKEY Update(const unsigned char RANDWIKEY[7],
9
                       const unsigned char PBID[4])
10
11
       int i,offset 1,offset 2;
12
13
       for (i = 0; i < 4; i++)
14
          LFSR[i] = RANDWIKEY[i+3] ^ WIKEY[i] ^ WIKEY[i+4];
15
       if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
16
          for (i = 0; i < 4; i++)
17
             LFSR[i] = RANDWIKEY[i+3];
18
       for (i = 0; i < 8; i++)
19
          Register[i] = WIKEY[i];
20
       Register[8] = AAV;
21
       for (i = 0; i < 3; i++)
22
          Register[i+9] = RANDWIKEY[i];
23
       for (i = 0; i < 4; i++)
24
          Register[i+12] = PBID[i];
25
       offset 1 = offset 2 = 128;
26
       CAVE(8, &offset 1, &offset 2);
27
       for (i = 0; i < 8; i++)
28
          WIKEY NEW[i] = Register[i] ^ Register[i+8];
29
    }
30
```

2.7.3. Wireline Interface Authentication Signature Calculation **Procedure**

3	Procedure name:	
4	WI_Auth_Signature	
5	Inputs from calling process:	
6	RAND_CHALLENGE 32 bits	
7	PBID - 30 bits	
8	ACRE_PHONE_NUMBER 24 bits	
9	Inputs from internal stored data:	
10	WIKEY 64 bits	
11	AAV 8 bits	
12	Outputs to calling process:	
13	AUTH_SIGNATURE 18 bits	
14	Outputs to internal stored data:	
15	None.	
16	This procedure is used to calculate 18-bit signatures used for verifying	ıg
17	WIKEY values.	
18	The initial loading of CAVE for calculation of wireline interface	сe
19	authentication signatures is given in Exhibit 2-38.	
20	For authentication of an ACRE, RAND CHALLENGE is receive	ed
21	from the PB as RAND_ACRE.	
22	For authentication of a PB, RAND CHALLENGE is received from the	ne.
23	ACRE as RAND_PB.	
24	The ACRE PHONE NUMBER is 24 bits comprised of the least	st
25	significant 24 bits of the ACRE's directory number (4 bits per digit	
26	The digits 1 through 9 are represented by their 4-bit binary value	ie
27	(0001b - 1001b), while the digit 0 is represented by 1010b. If the	
28	phone number of the acre is less than 6 digits, then the digits are fille	
29	on the left with zeros until 6 full digits are reached. Example: If the	
30	acre's phone number is (987) 654-3210, ACRE_PHONE_NUMBER is 010101000011001000011010b. If the earl's phone number is 8600	
31 32	010101000011001000011010b. If the acre's phone number is 8695	١,

Exhibit 2-38 CAVE Initial Loading for Wireline Interface Authentication Signatures

CAVE Item	Source Identifier	Size (Bits)
LFSR	RAND_CHALLENGE	32
Reg [0-7]	WIKEY	64
Reg [8]	AAV	8
Reg [9-11]	24 LSBs of ACRE_PHONE_NUMBER	24
Reg [12] 2 MSBs	00	2
Reg [12] 6 LSBs	6 MSBs of PBID	6
Reg [13-15]	24 LSBs of PBID	24

CAVE is run for eight rounds. The 18-bit result is AUTH_SIGNATURE. Exhibit 2-39 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-40.

The LFSR will initially be loaded with RAND_CHALLENGE. This value will be XOR'd with the 32 most significant bits of WIKEY XOR'd with the 32 least significant bits of WIKEY, then reloaded into the LFSR. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be reloaded with RAND_CHALLENGE to prevent a trivial null result.

The 18-bit authentication result AUTH_SIGNATURE is obtained from the final value of CAVE registers R00, R01, R02, R13, R14, and R15. The two most significant bits of AUTH_SIGNATURE are equal to the two least significant bits of R00 XOR R13. The next eight bits of AUTH_SIGNATURE are equal to R01 XOR R14. Finally, the least significant bits of AUTH SIGNATURE are equal to R02 XOR R15.

Exhibit 2-39 Calculation of AUTH SIGNATURE

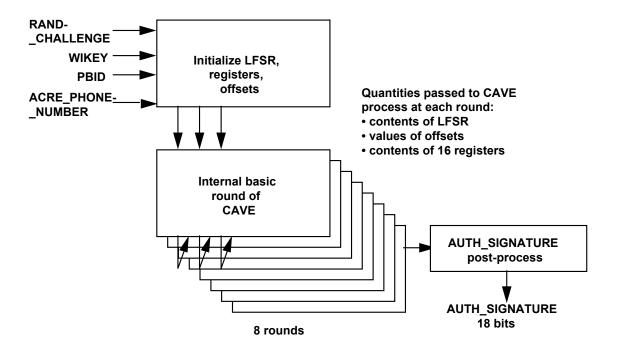


Exhibit 2-40 Code for calculation of AUTH SIGNATURE

```
/* WI Auth Signature has the same header as WIKEY Generation (see Exhibit 2-34) */
2
3
    /* Note that PBID is right justified. This means that the 2 MSBs of PBID
4
        must be set to 0 by the calling routine. */
5
6
    unsigned long WI Auth Signature (const unsigned char RAND CHALLENGE [4],
8
                                          const unsigned char PBID[4],
9
                                          const unsigned char ACRE PHONE NUMBER[3])
10
        int i,offset_1,offset_2;
11
        unsigned long AUTH SIGNATURE;
12
13
        for (i = 0; i < 4; i++)
14
           LFSR[i] = RAND_CHALLENGE[i] ^ WIKEY[i] ^ WIKEY[i+4];
15
        if ((LFSR[0] \mid LFSR[1] \mid LFSR[2] \mid LFSR[3]) == 0)
16
           for (i = 0; i < 4; i++)
17
               LFSR[i] = RAND_CHALLENGE[i];
18
        for (i = 0; i < 8; i++)
19
20
           Register[i] = WIKEY[i];
        Register[8] = AAV;
21
        for (i = 0; i < 3; i++)
22
           Register[i+9] = ACRE PHONE NUMBER[i];
23
        for (i = 0; i < 4; i++)
24
           Register[i+12] = PBID[i];
25
        offset_1 = offset_2 = 128;
26
        CAVE(8, &offset_1, &offset 2);
27
        AUTH SIGNATURE =
28
              ((unsigned long)(Register[0] ^ Register[13]) << 16) +
((unsigned long)(Register[1] ^ Register[14]) << 8) +
((unsigned long)(Register[2] ^ Register[15]))</pre>
29
30
31
32
           & 0x3ffff;
33
        return(AUTH SIGNATURE);
```

10

11

12

13

14

15

16

17

18

19

20

2.7.4. Wireless Residential Extension Authentication Signature Calculation Procedure

Procedure name:

WRE Auth Signature

Inputs from calling process:

RAND_WRE 19 bits ESN 32 bits PBID 30 bits

Inputs from internal stored data:

WRE_KEY 64 bits AAV 8 bits

Outputs to calling process:

AUTH SIGNATURE 18 bits

Outputs to internal stored data:

None.

This procedure is used to calculate 18-bit signatures used for verifying a mobile station.

The initial loading of CAVE for calculation of wireless residential extension authentication signatures is given in Exhibit 2-41.

Exhibit 2-41 CAVE Initial Loading for Residential Wireless Extension Authentication Signature

CAVE Item	Source Identifier	Size (Bits)
LFSR 19 MSBs	RAND_WRE	19
LFSR 13 LSBs	13 LSBs of PBID	13
Reg [0-7]	WRE_KEY	64
Reg [8]	AAV	8
Reg [9] 2 MSBs	00b	2
Reg [9] 6 LSBs	6 MSBs of PBID	6
Reg [10-11]	bits 23-8 of PBID	16
Reg [12-15]	ESN	32

22

23 24 CAVE is run for eight rounds. The 18-bit result is AUTH_SIGNATURE. Exhibit 2-42 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-43.

8

14

The 19 MSBs of LFSR will initially be loaded with RAND WRE. The 13 LSBs of LFSR will initially be loaded with the 13 LSBs of PBID. LFSR will be XOR'd with the 32 most significant bits of WRE KEY XOR'd with the 32 least significant bits of WRE KEY, then reloaded into the LFSR. If the resulting bit pattern fills the LFSR with all zeroes, then the 19 MSBs of LFSR will be reloaded with RAND WRE, and the 13 LSBs of LFSR will be reloaded with the 13 LSBs of PBID.

The 18-bit authentication result AUTH SIGNATURE is obtained from the final value of CAVE registers R00, R01, R02, R13, R14, and R15. The two most significant bits of AUTH_SIGNATURE are equal to the two least significant bits of R00 XOR R13. The next eight bits of AUTH SIGNATURE are equal to R01 XOR R14. Finally, the least significant bits of AUTH SIGNATURE are equal to R02 XOR R15.

Exhibit 2-42 Calculation of AUTH SIGNATURE

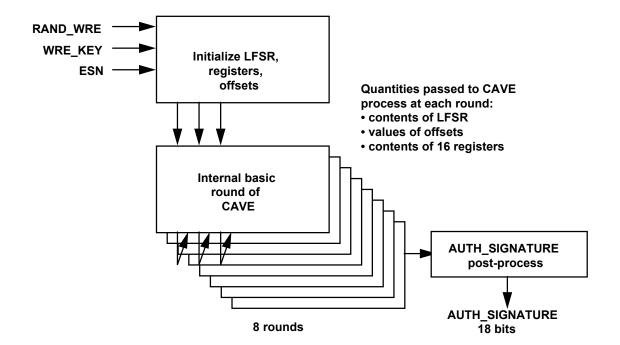


Exhibit 2-43 Code for calculation of AUTH SIGNATURE

```
/* WRE Auth Signature has the same header as WIKEY Generation (see
2
    Exhibit 2-34) */
3
4
    /* Note that RAND WRE is left justified and PBID is right justified.
5
       This means that the 5 LSBs of RAND WRE and the 2 MSBs of PBID
6
       must be set to 0 by the calling routine. */
7
8
9
    unsigned long WRE Auth Signature (const unsigned char RAND WRE[3],
10
                                         const unsigned char PBID[4],
                                         const unsigned char ESN[4])
11
12
       int i,offset_1,offset_2;
13
       unsigned long AUTH SIGNATURE;
14
15
       for (i = 0; i < 3; i++)
16
          LFSR[i] = RAND_WRE[i];
17
       LFSR[2] = LFSR[2] | (PBID[2] & 0x1F);
18
       LFSR[3] = PBID[3];
19
       for (i = 0; i < 4; i++)
20
          LFSR[i] = LFSR[i] ^ WRE_KEY[i] ^ WRE_KEY[i+4];
21
       if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
22
23
          for (i = 0; i < 3; i++)
24
             LFSR[i] = RAND WRE[i];
25
          LFSR[2] = LFSR[2] (PBID[2] & 0x1F);
26
          LFSR[3] = PBID[3];
27
28
       for (i = 0; i < 8; i++)
29
          Register[i] = WRE KEY[i];
30
       Register[8] = AAV;
31
32
       for (i = 0; i < 3; i++)
          Register[i+9] = PBID[i];
33
       for (i = 0; i < 4; i++)
34
           Register[i+12] = ESN[i];
35
       offset_1 = offset_2 = 128;
36
       CAVE(8, &offset_1, &offset 2);
37
       AUTH SIGNATURE =
38
             ((unsigned long)(Register[0] ^ Register[13]) << 16) +
((unsigned long)(Register[1] ^ Register[14]) << 8) +</pre>
39
40
              ((unsigned long)(Register[2] ^ Register[15]))
41
           & 0x3ffff;
42
       return(AUTH SIGNATURE);
43
44
45
```

2.8. Basic Wireless Data Encryption

Data encryption for wireless data services is provided by the ORYX algorithm (as named by its developers) which is described in the following.

The DataKey Generation Procedure uses the A, B, and K registers
to generate a DataKey. SSD_B provides the sole input to this
procedure. If the data encryptor has access to SSD_B, DataKey
may be generated locally. If not, DataKey is calculated elsewhere,
then sent to the encryptor.

In the network, this procedure executes at the initial serving system if SSD_B is shared or at the authentication center if SSD_B is not shared. DataKey may be precomputed when the mobile station registers.

- The LTable Generation Procedure uses the K register to generate a lookup table. RAND provides the sole input to this procedure. L is generated locally. In the network, this procedure executes at the initial serving system, and after intersystem handoff, it may execute at subsequent serving systems.
- The Data_Mask Procedure provides an encryption mask of the length requested by the calling process. It uses four inputs:
 - DataKey from the DataKey Generation Procedure via the calling process;
 - 2. HOOK directly from the calling process;
 - 3. len directly from the calling process; and
 - 4. L as stored from the LTable Generation Procedure.

The encryption mask is generated locally.

ORYX uses 3 Galois shift registers: A, B, and K. ORYX also uses a 256-octet look up table L.

Register K is a 32-bit Galois shift register, with feedback polynomial

$$k(z) = z^{32} + z^{28} + z^{19} + z^{18} + z^{16} + z^{14}$$

+ $z^{11} + z^{10} + z^9 + z^6 + z^5 + z + 1$.

This is implemented by shifting the contents of K to the right and XORing the bit shifted out of the right-most position into the bit positions specified by the feedback polynomial.

Before stepping, a check is made to see if all of the bit positions in K are zero. If they are, K is initialized with the hex constant 0x31415926.

1

2

3

4

6

8

10

11

12 13

15

16

17

18

19

20

21

22

23

24

25

26

26

The feedback polynomial k(z) is primitive and has Peterson & Weldon octal code 42003247143.1) 2 Registers A and B are 32 bit Galois shift registers, shifting to the left: the leftmost bit is XORed into the bit positions specified by the feedback polynomial. Register A sometimes steps with feedback 5 polynomial 6 $a_1(z) = z^{32} + z^{26} + z^{23} + z^{22} + z^{16} + z^{12} + z^{11}$ 7 $+z^{10}+z^8+z^7+z^5+z^4+z^2+z+1$ 8 and sometimes with feedback polynomial $a_2(z) = z^{32} + z^{27} + z^{26} + z^{25} + z^{24} + z^{23} + z^{22} + z^{17}$ 10 $+z^{13}+z^{11}+z^{10}+z^9+z^8+z^7+z^2+z+1$ 11 The decision is based on the current high order bit of K. First K is 12 stepped. If the (new) high order bit of K is set, register A steps 13 according to polynomial a₁(z); if the high order bit of K is clear, register A steps according to polynomial $a_2(z)$. 15 Register B steps once if the next-to-high order bit of K is clear, or twice 16 if the next-to-high order bit of K is set, with feedback polynomial 17 $b(z) = (z+1)(z^{31} + z^{20} + z^{15} + z^5 + z^4 + z^3 + 1)$ = $z^{32} + z^{31} + z^{21} + z^{20} + z^{16} + z^{15} + z^6 + z^3 + z + 1$ 18 19 This is also implemented with a left shift, XORing the leftmost bit into 20 the bit positions specified by the feedback polynomial. 21 Polynomials $a_1(z)$, $a_2(z)$, and the degree 31 factor of polynomial b(z)22 are all primitive, with Peterson & Weldon octal codes 40460216667, 23 41760427607, and 20004100071, respectively. 24

used in ORYX.

1) Since each shift register always has its output connected to its feedback gates, the most-significant bit is not required explicitly in the accompanying C code, hence the leading 4 (octal) is omitted from the representations of the polynomial within the C code.

Exhibit 2-44 illustrates the operation of the three Galois shift registers

Exhibit 2-44 Galois Shift Registers

Exhibit 2-44 Galois Shift Registers				
	0		回压木	ste r
			1 32	Bit Nbr Register K
	0 5		E 0 4	0
gate (AND ga	- 0 0 3	F -Q= -	0 30	- 0 0
ž d		₽ - © - -	0 0	0 0
Fe echack gate (AND gate)	4	—— —	28 - 2	ω -
	r - 0 0	<u> </u>	0 0	4 0 -
	9 0 0 7	<u> </u>	0 58	· 0
	r - 0 - 0 0 -	₩	0 0 22	9 0
	8 0	₩	0 24	· 0 0
	6 0 - 0		0 23	8 0
	0 - 2 - 5 0	□	0 0 5	o 0
	0 7 - 7	⊕	0 0	0 0 0
	- 0 0		0 50	0 7
	0 2 0 0		φ – κ	- 7
	4 0 0 0		8 -	3 - 13
	5 0 0 -	. 1 — — — — — — — — — — — — — — — — — —	0 0	4 0
, g — — — — — — — — — — — — — — — — — —	9 - 0 0 4 - E	K Register	\rightarrow	£
A and B Registers	0 - 0	* * * *	\longrightarrow 11	
	8 0 0 0		12 0	
	5 00004		4 +	1 17
A and 6	0 0 50		£ 0 4	0 0
	7 0 0 7		0 72	6 0 4
	8 - 0 - 0 -		<u>+</u>	7 20
	0 - 2		1 1 2	- 5
	25 0 1 0		σ ←	23 - 53
	52 0 4 - 1- 0 0		8 0	0 23
	7 - 6		7 0 1	0 0
	0 1 0		9 -	72 - 9
× × × × × × × × × × × × × × × × × × ×	87 0 0 0 1 0 0		· σ ←	7 28
Je or	0 0 0 0		4 0 4	0
Shift register Exclusive OR (XOR)	8 0 0 0		e 0	0 0 0
Shift register Exclusive OR (XOR)	2 - 1 0 0 0 0 3	F-0-	0	0 0
	000000 00000 00000		— — დ	1 30
	Bit Nbr Regiser A ₁ Octal A ₂		0 -	<u>8</u> – 8
			6000000	COCCUSED Y

Information disclosed in this document is subject to the export jurisdiction of the US Department of Commerce as specified in Export Administration Regulations (title 15 CFR parts 730 through 774 inclusive).

81

2.8.1. Data Encryption Key Generation Procedure

2	Procedure name:	
3	DataKey_Generation	
4	Inputs from calling process:	
5	None.	
6	Inputs from internal stored data:	
7	SSD_B	64 bits
8	Outputs to calling process:	
9	DataKey	32 bits
10	Outputs to internal stored data:	
11	None.	
	Trong.	
12 13	This procedure generates DataKo procedure (see 2.8.3).	ey, a key used by the Data_Mask
14	The calculation of DataKey de	enends only on SSD B therefore
15	The calculation of DataKey depends only on SSD_B, therefore DataKey may be computed at the beginning of each call using the	
16	current value of SSD_B, or it may be computed and saved when SSD is	
17	updated. The value of DataKey sha	all not change during a call.
18	Here is how DataKev is formed fi	rom SSD B, using ORYX as a hash
19		ized with the first 32 bits of SSD_B,
20		32 bits of SSD_B and K is initialized
21	with the XOR of A and B. Then K	is stepped 256 times.
22	After the i-th step, for $0 \le i < 250$	6, the i-th entry, L[i], in the look up
23		t-significant octet of K. Then the
24	following three-step procedure is re	epeated 32 times:
25	1. ORYX is stepped by calling the	he keygen() procedure, producing
26	a key octet, which is tempor	rarily stored in the variable temp.
27		fting its contents to the left by 9 bits
28	and adding the contents of ter	mp.
29		g a key octet, which is temporarily
30		Register B is modified by shifting its
31	•	d adding the contents of temp.
32		g a key octet, which is temporarily
33	used to modify K as described	The value of the variable temp is in Exhibit 2-45
35	The XOR of the final values of K,	
	THE ACK OF the fillar values of K,	rs, and D is stored in Datakey.
36	Exhibit 2-45 describes the calculati	ion in ANSI C.

Exhibit 2-45 Header for Basic Data Encryption

```
unsigned long DataKey Generation(void);
2
3
4
    void LTable Generation(const unsigned char [] );
5
    void Data_Mask(const unsigned long ,
6
                    const unsigned long ,
8
                    const int ,
9
                    unsigned char []);
10
11
    #ifndef ORYX SOURCE FILE
    extern
12
    unsigned char L[256];
13
    extern
14
    unsigned long DataKey;
15
    #endif
16
17
                              Exhibit 2-46 DataKey Generation
18
    #define ORYX SOURCE FILE
19
    #include "cavei.h" /* see Exhibit 2-3 */
#include "oryx.h" /* see Exhibit 2-45 */
20
21
22
    \#define high(x) (unsigned char) (0xffU&(x>>24)) /* leftmost octet */
23
    \#define FA1 000460216667 /* Peterson & Weldon prim 32 */
                                      /* Peterson & Weldon prim 32 */
25
    #define FA2 001760427607
    #define FB 020014300113
                                     /* P&W prim 31 020004100071 times z+1 */
26
                                      /* reverse of P&W prim 32 042003247143
    #define FK 030634530010
28
29
   static
    unsigned long K;
                                      /* 32-bit K register */
   static
32
    unsigned long A, B;
                                      /* 32-bit LFSRs */
33
34
    unsigned char L[256];
                                     /* look up table */
35
    unsigned char L[256];
unsigned long DataKey;
36
                                      /* data encryption key */
37
38
   static
    void kstep(void);
39
40
    static
    unsigned char keygen (void);
41
```

```
unsigned long DataKey Generation(void)
1
2
        int i;
3
       unsigned long temp;
6
       A = 0;
        for (i=0; i<4; i++)
7
           A = (A << 8) + (unsigned long)SSD B[i];
8
        B = 0;
9
        for (i=4; i<8; i++)
10
           B = (B << 8) + (unsigned long)SSD B[i];
11
12
        K = A ^ B;
13
        for(i=0; i<256; i++)
14
15
           kstep();
16
           L[i] = high(K);
17
18
        for(i=0; i<32; i++)
19
20
           temp = (unsigned long)keygen();
21
           A = (A << 9) + temp;
22
           temp = (unsigned long)keygen();
23
           B = (B << 9) + temp;
24
           temp = (unsigned long)keygen();
25
26
           K = (0xff00ffffU \& K) + (temp << 16);
           K &= 0xffff00ffU + (temp<<8);</pre>
27
28
        return ( (A ^ B ^ K) & Oxffffffff );
29
30
31
    static
32
    unsigned char keygen(void)
33
34
       unsigned char x;
35
       int i, trips;
36
37
38
       kstep();
39
         * if high bit of K set, use A1 feedback
40
         * otherwise use A2 feedback
41
         * /
42
        if((1UL<<31) & A)
43
44
           A += A;
45
           if((1UL<<31) & K)
46
              A = A ^ FA1;
47
           else
48
              A = A ^ FA2;
49
50
        else
51
           A += A;
52
```

```
1
         * if next-high bit of K set, step B twice
         * otherwise once
3
         * /
5
        if((1UL<<30) & K)
           trips = 2;
6
7
        else
           trips = 1;
8
        for(i=0; i<trips; i++)</pre>
9
10
            if((1UL<<31) & B)
11
12
               B += B;
13
               B = B \hat{B};
14
15
            else
16
               B += B;
17
18
        \dot{x} = high(K) + L[high(A)] + L[high(B)];
19
                                                      /* use only 8 bits */
        x \&= 0xffU;
20
21
        return x;
22
23
24
     * step the K register
25
26
     */
27
    static
    void kstep(void)
28
29
        if(K==0) K = 0x31415926;
30
        if(K&1)
31
32
           K = (K >> 1) ^ FK;
33
34
        else
35
36
37
           K = (K >> 1);
38
        K &= Oxfffffff;
39
40
41
```

2.8.2. L-Table Generation Procedure

2	Procedure name:	
3	LTable_Generation	
4	Inputs from calling proces	SS:
5	RAND	32 bits
6	Inputs from internal stored	d data:
7	None.	
8	Outputs to calling process	:
9	None.	
10	Outputs to internal stored	data:
11	L	256*8 bits
12 13	This procedure generates (see 2.8.3).	L, a table used in the Data_Mask procedure
14 15 16 17	each call, and may be each	procedure shall be executed at the beginning of executed after intersystem handoff, using the at the start of the call. The value of L shall not
18	L is initialized as follows:	
19	K is set equal to RAN	ND.
20 21	The i-th cell in the L \leq i \leq 256.	table, L[i], is initialized with the value i, for 0
22 23 24 25	\leq i $<$ 256, the value	s stepped 256 times. After the i-th step, for 0 e stored in the cell whose index is the most and the value stored in the i-th cell of the L d.
26	Exhibit 2-47 describes the	e calculation in ANSI C.

Exhibit 2-47 LTable Generation

```
2
    /* The header for LTable Generation is the same as for
       DataKey Generation (see Exhibit 2-46).*/
3
4
    void LTable_Generation(const unsigned char RAND[4])
5
6
7
       int i,j;
8
       unsigned char tempc;
9
10
       K = 0;
       for(i=0; i<4; i++)
11
          K = (K << 8) + (unsigned long)RAND[i];
12
       for (i=0; i<256; i++)
13
          L[i] = (unsigned char)i;
14
15
       /* use high octet of K to permute 0 through 255 */
16
       for (i=0; i < 256; i++)
17
18
          kstep();
19
20
          j = high(K);
21
          tempc = L[i];
          L[i] = L[j];
          L[j] = tempc;
23
24
25
```

2.8.3. Data Encryption Mask Generation Procedure

2	Procedure name:	
3	Data_Mask	
4	Inputs from calling process:	
5	DataKey	32 bits
6	HOOK	32 bits
7	len	integer
8	Inputs from internal stored data:	
9	L	256*8 bits
10	Outputs to calling process:	
11	mask	len*8 bits
12	Outputs to internal stored data:	
13	None.	
13	rone.	
14	This procedure generates an encry	ption mask of length len*8.
15	Implementations using data encry	ption shall comply with the following
16		s apply to all data encrypted during a
17	call.	
18	• The least-significant bits of H	OOK shall change most frequently.
19	A mask produced using a second control of the second control	value of HOOK should be used to
20	encrypt only one set of data.	value of frook should be used to
21	A mask produced using a vi	alue of HOOK shall not be used to
22	 A mask produced using a value of HOOK shall not be used to encrypt data in more than one direction of transmission, nor shall it 	
23	be used to encrypt data on mo	
24	The DataKey and the look up :	table L must be computed prior to
25	executing Data_Mask.	more 2 must be computed prior to
	5	
26	The key octets in a frame mask are	e produced by initializing the registers
27		om DataKey and HOOK as follows.
28	1. K is set equal to the current v	alue of HOOK. If K_1 , K_2 , K_3 , and K_4
29		he following assignments are made in
30	turn:	

```
1 K_{2} = L[K_{2} + K_{4}] K_{3} = L[K_{3} + K_{4}] K_{4} = L[K_{4}]
```

11

where the additions $K_i + K_4$ are performed modulo 256.

- 2. K is stepped once, and A is set equal to DataKey XOR-ed with K.
- 3. K is stepped again, and B is set equal to DataKey XOR-ed with K.
- 4. K is stepped again, and K is set equal to DataKey XOR-ed with K.

With these values of A, B, and K, the ORYX key generator is stepped n times, and the resulting key octets are the n octets of the frame mask.

Exhibit 2-48 describes the calculation in ANSI C.

Exhibit 2-48 Data Encryption Mask Generation

```
12
    /* Data Mask has the same header as DataKey Generation
13
        (see Exhibit 2-46) */
14
    void Data Mask (const unsigned long DataKey,
15
                     const unsigned long HOOK,
16
                     const int len.
17
18
                     unsigned char mask[] )
19
        int i;
20
21
        K = (unsigned long)L[HOOK&0xff];
22
23
        K += ((unsigned long) L[((HOOK>>8) + HOOK) & 0xff]) << 8;
        K += ((unsigned long)L[((HOOK>>16)+HOOK)&0xff])<<16;
24
25
        K += ((unsigned long)L[((HOOK>>24)+HOOK)&0xff])<<24;
       kstep(); A = DataKey ^ K; /* kstep() is defined in Exhibit 2-45 */
kstep(); B = DataKey ^ K;
26
27
       kstep(); K = DataKey ^ K;
28
29
        for(i=0; i<len; i++)</pre>
30
           mask[i] = keygen(); /* keygen() is defined in Exhibit 2-45 */
31
32
```

2.9. Enhanced Voice and Data Privacy

This section defines key generation and encryption procedures for the following TDMA content: voice, DTC and DCCH messages, and RLP data.

There are three key generation procedures: DTC key schedule generation, DCCH key schedule generation, and a procedure that each of these call termed the SCEMA Secrets Generation. The DCCH key schedule is based on a CMEA Key instance which is generated at Registration and remains for the life of the Registration. The DTC key is generated from the CMEA Key on a per call basis.

The encryption procedures contained herein are grouped into three levels, where the higher level procedures typically call procedures from a lower level. Level 1 has one member: the SCEMA encryption algorithm. Level 2 contains three procedures: a Long Block Encryptor for blocks of 48 bits, a Short Block Encryptor for blocks less than 48 bits, and a KSG used in voice and message encryption. Level 3 contains voice, message, and RLP data encryption procedures which interface directly to TIA/EIA-136-510.

CAVE algorithm code used in this section but defined external to it comprises CAVE header files, "cave.h" (see Exhibit 2-2) and "cavei.h" (see Exhibit 2-3), and CAVE source code (see Exhibit 2-4).

Throughout this section, the source code exhibits will be tagged with file names. While these names are arbitrary, they serve as a visual aid to the reader to flag a source code file and differentiate it from header files.

2.9.1. SCEMA Key Generation Code

This section describes the procedures used for generating secret key schedules for use in Enhanced Privacy and Encryption (EPE). Separate schedules are generated for the TDMA DTC (Digital Traffic Channel) and the DCCH (Digital Control Channel).

2.9.1.1. DTC Key Generation

Procedure name:

DTC_Key_Generation

Inputs from calling process:

None.

Inputs from internal stored data:

CMEA Key (implicitly)

Outputs to calling process:

None.

Outputs to internal stored data:

DTC key schedule structure

This procedure creates an array of DTC key schedule structures. Currently, the array contains a single element but allows the option to be extended in the future to accommodate multiple key schedules of different strengths. Each array element is a structure containing *scemaKey, *obox, *offKey, and neededLength The first three elements are pointers to keys (cryptovariables). The fourth, called neededLength, generally corresponds to the true entropy of the key, and is set in "scema.h" (see Exhibit 2-53).

dtcScheds[0] is generated from the CMEA Key. In TIA/EIA-136-510, this 45-octet schedule is termed DTCKey. These 45 octets comprise

dtcScemaKeyCK18 octetsdtcOboxCK132 octetsdtcOffKeyAuxCK14 octetsNeededLengthCK11 octet

The suffix "CK1" denotes CaveKey1.

Exhibit 2-49 SCEMA DTC Key Generation

```
2
    /* SCEMA DTC Key Generation "dtcKeyGen.c" */
3
4
    #include "scema.h" /* see Exhibit 2-53 */
6
8
    dtcScheds[0] accesses DTC CaveKey1 schedule.
9
10
    unsigned char dtcScemaKeyCK1[ScemaKeyLengthCK1];
11
    unsigned int dtcOboxCK1[16];
12
    unsigned int
                   dtcOffKeyAuxCK1[2];
13
14
    keySched dtcScheds[] = {
15
       {dtcScemaKeyCK1, dtcOboxCK1, dtcOffKeyAuxCK1, NeededLengthCK1},
16
17
18
19
20
    void DTC Key Generation(void)
21
       SCEMA Secret Generation(dtcScheds);
22
23
24
25
26
    Note: If a key schedule of a different strength is required in the
27
    the following can serve as an example:
28
29
30
31
    dtcScheds[0] will access DTC CaveKey1 schedule.
    dtcScheds[1] will access DTC TBD Key2 schedule.
32
33
34
    unsigned char dtcScemaKeyCK1[ScemaKeyLengthCK1];
35
    unsigned int dtcOboxCK1[16];
36
37
    unsigned int
                   dtcOffKeyAuxCK1[2];
38
    unsigned char dtcScemaKeyTbdK2[ScemaKeyLengthTbdK2];
39
    unsigned int
                    dtcOboxTbdK2[16];
40
    unsigned int
                    dtcOffKeyAuxTbdK2[2];
41
42
    keySched dtcScheds[] = {
43
        {dtcScemaKeyCK1, dtcOboxCK1, dtcOffKeyAuxCK1, NeededLengthCK1},
44
        dtcScemaKeyTbdK2, dtcOboxTbdK2, dtcOffKeyAuxTbdK2,
45
46
    NeededLengthTbdK2 }
47
    };
48
    * /
49
```

2.9.1.2. DCCH Key Generation

25

27

28

Procedure name: DCCH Key Generation Inputs from calling process: None. Inputs from internal stored data: CMEA Key (implicitly) Outputs to calling process: None. 10 Outputs to internal stored data: 11 dcchScheds[] DCCH key schedule structure 12 13 14 This procedure creates an array of DCCH key schedule structures. Currently, the array contains a single element but allows the option to 16 be extended in the future to accommodate multiple key schedules of 17 different strengths. Each array element is a structure containing 18 *scemaKey, *obox, *offKey, and neededLength The first three 19 elements are pointers to keys (cryptovariables). The fourth, called 20 neededLength, generally corresponds to the true entropy of the key, and 21 is set in "scema.h" (see Exhibit 2-53). 22 dcchScheds[0] is generated from the CMEA Key. In TIA/EIA-136-510, 23 this 45-octet schedule is termed DCCHKey. These 45 octets comprise 24

dcchScemaKeyCK18 octetsdcchOboxCK132 octetsdcchOffKeyAuxCK14 octetsNeededLengthCK11 octet

The suffix "CK1" denotes CaveKey1.

Exhibit 2-50 SCEMA DCCH Key Generation

```
/* SCEMA DCCH Key Generation "dcchKeyGen.c" */
2
3
    #include "scema.h" /* see Exhibit 2-53 */
4
6
    dcchScheds[0] accesses DCCH CaveKey1 schedule.
7
8
10
    unsigned char dcchScemaKeyCK1[ScemaKeyLengthCK1];
    unsigned int dcchOboxCK1[16];
11
    unsigned int
                   dcchOffKeyAuxCK1[2];
12
13
    keySched dcchScheds[] = {
14
       {dcchScemaKeyCK1, dcchOboxCK1, dcchOffKeyAuxCK1, NeededLengthCK1},
15
16
17
18
    void DCCH Key Generation(void)
19
20
21
       SCEMA Secret Generation(dcchScheds);
22
23
24
25
    Note: If a key schedule of a different strength is required in the
26
27
    see the example in dtcKeyGen.c.
28
29
30
```

2.9.1.3. SCEMA Secret Generation 2 Procedure name: SCEMA Secret Generation Inputs from calling process: None. Inputs from internal stored data: CMEAKEY[0-7] 64 bits 8 Outputs to calling process: 9 None. 10 Outputs to internal stored data: 11 SCEMA KEY [0-7] 64 bits 12 oboxSchedFin[0-15] 16 words (256 bits) 13 offKeyAuxFin[0-1] 2 words (32 bits) 14 15 The CMEA Encryption Key and VPM Generation Procedure, defined 16 in section 2.5.1, is used to generate a CMEA key on a per-call basis. 17 SCEMA requires additional secret values to be generated on a per-call or per-registration basis. This procedure accomplishes this by running 19 the CAVE algorithm initialized by the original CMEA key (64 bits). 20 First, the LFSR will be loaded with the 32 MSBs of the CMEA 21 key. If these MSBs are all zero, then a constant, 0x31415926, will be loaded instead. 23 Second, registers R00 through R07 will be loaded with the 24 CMEA key. 25 Third, registers R08 through R15 will be loaded with the one's-26 complement of the CMEA key. 27 Fourth, the offset table pointers will be reset to all zeros. 28 Fifth, the LFSR is loaded before all of the remaining iterations 29 with a "roll-over RAND" comprised of the contents of R00, 30 R01, R14, and R15 at the end of the previous iteration. If the 31 resulting bit pattern fills the LFSR with all zeros, then the LFSR 32 will be loaded with the constant, 0x31415926. 33 The SCEMA key octets are drawn as follows (assuming that none 34 equate to zero): 35 scema key[0] = register[4] XOR register[8]; (iteration 2)scema key[1] = register[5] XOR register[9]; (iteration 2) 37

38

scema key [2] = register[6] XOR register[10]; (iteration 2)

1	• scema_key[3] = register[7] XOR register[11]; (iteration 2)
2	• scema_key[4] = register[4] XOR register[8]; (iteration 3)
3	• scema_key[5] = register[5] XOR register[9]; (iteration 3)
4	• scema_key[6] = register[6] XOR register[10]; (iteration 3)
5	• scema_key[7] = register[7] XOR register[11]; (iteration 3)
6	
7	Note: If, during this process, any of the octets of SCEMA KEY as
8	defined above are zero, that octet is replaced by the next nonzero octet
9	generated. Additional iterations are performed as necessary to generate
10	eight nonzero octets for SCEMA_KEY. Thus the output of the CAVE
11	iterations can be viewed as SCEMA_KEY candidates which are then
12	screened to yield the actual SCEMA_KEY.
13	The Obox table comprises 16 16-bit words. Its values are drawn in a
14	similar manner with the following exceptions: First, the LSB and MSB
15	octets of the words are filled in succession. Second, a different screen is
16	used here which rejects those Obox table candidates where the 4 LSBs
17	of the sum of the table values and its index equals zero.
18	Finally, the two auxiliary offset keys are derived as follows via a single
19	CAVE iteration:
20	• offKeyAuxFin[0] (lower octet) = register[4] XOR register[8]
21	• offKeyAuxFin[0] (upper octet) = register[5] XOR register[9]
22	• offKeyAuxFin[1] (lower octet) = register[6] XOR register[10]
23	• offKeyAuxFin[1] (upper octet) = register[7] XOR register[11]

Exhibit 2-51 Generation of SCEMA Secrets

2

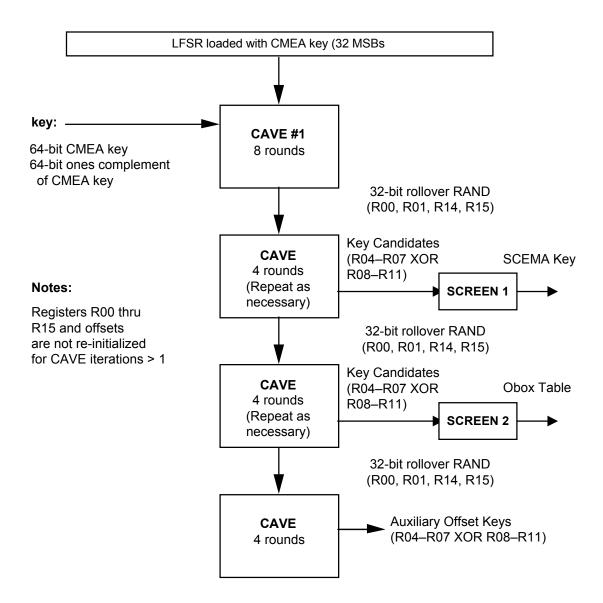


Exhibit 2-52 SCEMA Secret Generation

```
/* SCEMA Financial Secret Generation has the same header as SCEMA (see
2
    Exhibit 2-53) */
3
    /* SCEMA Secret Generation "scemaKeyGen.c */
4
    #include "cavei.h" /* see Exhibit 2-3 */
6
    #include "scema.h" /* see Exhibit 2-53 */
7
8
9
10
    /* CAVE-related code */
11
    void roll LFSR SCEMA(void)
12
13
       LFSR A = Register[0];
14
       LFSR B = Register[1];
15
       LFSR C = Register[14];
16
       LFSR D = Register[15];
17
18
       if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
19
20
21
           LFSR A = 0x31;
           LFSR_B = 0x41;
22
           LFSR_C = 0x59;
23
           LFSRD = 0x26;
24
25
26
    }
27
28
    void SCEMA Secret Generation(keySched *schedPtr)
29
30
31
       int i,j,offset 1,offset 2;
32
33
       /* iteration 1, first pass through CAVE */
34
       for (i = 0; i < 4; i++)
35
          LFSR[i] = cmeakey[i+4];
36
37
       if ((LFSR A | LFSR B | LFSR C | LFSR D) == 0)
38
39
           LFSR A = 0x31;
40
           LFSR^{-}B = 0x41;
41
           LFSR_C = 0x59;
42
43
           LFSR D = 0x26;
44
       for (i = 0; i < 8; i++)
45
46
          Register[i] = cmeakey[i];
47
       for (i = 8; i < 16; i++)
48
          Register[i] = ~cmeakey[i-8];
49
50
       offset_1 = 0x0;
51
       offset 2 = 0x0;
52
53
54
       CAVE(8, &offset 1, &offset 2);
55
```

```
/* Generation of SCEMA KEY */
1
2
       i = 0; j = 4;
3
       while (i < ScemaKeyLengthCK1)</pre>
5
           /* see if new key material needs to be generated */
6
          if(j == 4)
7
8
              j = 0;
9
              roll LFSR SCEMA();
10
              CAVE(4, &offset 1, &offset 2);
11
12
13
          schedPtr->scemaKey[i] = Register[j+4] ^ Register[j+8];
14
          j++;
15
16
          /* advance to next octet of SCEMA KEY if not zero; otherwise
17
              generate another value */
18
19
          if (schedPtr->scemaKey[i] != 0)
20
              i++;
21
       }
22
23
       /* Generation of SCEMA Obox Table */
24
25
       i = 0; j = 4;
26
       while (i < 16)
27
28
           /* see if new key material needs to be generated */
29
          if(j == 4)
30
31
              j = 0;
32
              roll LFSR SCEMA();
33
              CAVE(4, &offset 1, &offset 2);
34
35
36
           schedPtr->obox[i] =
37
              (int)(((Register[j+4] ^ Register[j+8]) & 0xFF) |
38
                     ((Register[j+5] ^ Register[j+9]) << 8));
39
           j += 2;
40
41
           /* advance to next octet of Obox Table if not zero; otherwise
42
              generate another value */
43
44
          if (((schedPtr->obox[i] + i) & 0x0F) != 0)
45
              i++;
46
47
48
       /* Generation of SCEMA auxiliary offset keys */
49
50
       roll LFSR SCEMA();
51
       CAVE(4, &offset 1, &offset 2);
52
53
       schedPtr->offKey[0] = (int)(((Register[4] ^ Register[8]) & 0xFF) |
54
                     ((Register[5] ^ Register[9]) << 8));
55
56
       schedPtr->offKey[1] = (int)(((Register[6] ^ Register[10]) & 0xFF) |
57
                     ((Register[7] ^ Register[11]) << 8));
58
59
60
61
```

2

5

52

2.9.2. SCEMA Header File

This section contains the header file used for all of the procedures in EPE. Some of the procedures additionally use CAVE header files, "cave.h" (see Exhibit 2-2) and "cavei.h" (see Exhibit 2-3).

Exhibit 2-53 SCEMA Header File

```
6
    /* SCEMA Header File "scema.h" */
8
    /* Key schedule architecture */
10
    typedef struct _key_sched {
11
       unsigned char *scemaKey;
12
       unsigned int *obox;
13
       unsigned int *offKey;
14
       unsigned char neededLength;
15
    } keySched;
16
17
18
    keySched dtcScheds[];
19
    keySched dcchScheds[];
20
21
22
23
    /* SCEMA procedure/function declarations */
24
    void DTC Key Generation(void);
25
26
    void DCCH Key Generation(void);
27
28
    void SCEMA Secret Generation(keySched *schedPtr);
29
30
31
    void SCEMA (unsigned char *msq buf,
                const int octet count,
32
                const unsigned char *csync,
33
34
              const unsigned char id,
              const unsigned char idMask,
35
                const unsigned int decrypt,
36
                keySched *schedPtr);
37
38
39
    void SCEMA_KSG(unsigned char *keystreamBuf,
                 const unsigned int requestedStreamLen,
40
                 const unsigned char *inputBuf,
41
                 const unsigned int inputLen,
42
43
                 const unsigned char contentType,
                 keySched *schedPtr,
44
45
                 const unsigned int direction);
46
    void Long Block Encryptor(unsigned char *contentBuf,
47
                           const unsigned char contentType,
48
                           const unsigned int decrypt,
49
                           keySched *schedPtr,
50
                           const unsigned int direction);
51
```

```
void Short Block Encryptor(unsigned char *contentBuf,
1
                    const unsigned int numBits,
2
                    const unsigned char contentType,
3
                    const unsigned char *entropy,
                          const unsigned int decrypt,
                          keySched *schedPtr,
6
                          const unsigned int direction);
8
    void Enhanced Message Encryption (unsigned char *msgBuf,
9
                          const unsigned int numBits,
10
                          const unsigned int dcchDTC,
11
                          const unsigned char *rand,
12
                          const unsigned char msgType,
13
                          const unsigned int decrypt,
14
                          const unsigned int keyGenerator,
15
                          const unsigned int direction);
16
17
    void Enhanced Voice Privacy(const unsigned int coderVer,
18
                          unsigned char *speechBuf1,
19
                          const unsigned int numlaBits,
20
                          unsigned char *speechBufRem,
21
                          const unsigned int numRemBits,
22
                          const unsigned int decrypt,
23
                          const unsigned int keyGenerator,
24
                          const unsigned int direction);
25
26
27
    void Enhanced Data Mask (unsigned char *mask,
                       const unsigned long HOOK,
28
                       const unsigned int len,
29
30
                       const unsigned int keyGenerator);
31
    /* Encryption mode of SCEMA */
32
33
    #define ENCRYPTING
34
    #define DECRYPTING
35
                          1
    /* Blocksize of plaintext (or ciphertext) */
37
    #define ThreeOctets
38
    #define SixOctets 6
39
    #define EightOctets
40
41
    /* Long Block Definitions
42
43
    Note: The LongBlockArchitecture identity segment forces a one into bit 2
    of SCEMA's cryptosync top octet to differentiate the Long Block
44
    Encryptor from all other KSG-type encryptors.
45
46
47
    #define LongBlkIdMask
48
                                    0xFF
49
    #define LongBlockArchitecture 0x04
50
    /* KSG, RLP, and Short Block Definitions
51
    Note: The LongBlockArchitecture identity segment forces a zero into bit
52
    2 of SCEMA's cryptosync top octet.
53
    * /
54
55
    #define KSGIdMask
    #define KSGArchitecture 0x00
57
    /* Content Types */
58
    #define VoiceContent 0x00
59
60
    #define MessageContent 0x10
    #define RlpContent
                             0x20
```

```
1
    /* Direction */
    #define ForwardChannel 1
3
    #define ReverseChannel 0
   /* Instances */
6
   #define Instance1 0x01
   #define Instance2 0x02
8
    #define Instance3 0x03
10
  /* DCCH/DTC */
11
   #define DCCH 0
12
    #define DTC 1
13
14
    /* Message Types */
15
   #define TestMsgType 0x1A
16
    #define TestMsgType2 0x09
17
18
    /* Used in SCEMA transforms */
19
    #define OFFSETA ((unsigned char)(*offInt & 0xFF))
20
    #define OFFSETB ((unsigned char)((*offInt >> 8) & 0xFF))
    /* Miscellaneous */
23
   #define MaxFrameOctetSize 35 /* 278 bits */
    #define MaxMessageOctetSize
                                   256 /* 2048 bits */
    #define CAVEKey1 1
    #define CoderVersionZero 0
27
   \#define MAX(A,B) ((A) > (B) ? (A) : (B))
29
    \#define MIN(A,B) ((A) < (B) ? (A) : (B))
30
31
    unsigned int offsetInt[2];
32
33
    /* Key length determination and individual key schedule architectures
34
    Note: NeededLength must be <= length of scemaKey to prevent
    stbox() overflow, and should be >= the key schedule entropy.
    Also, it must be even.
    If a key schedule of a different strength is required in the future,
39
    replicate the below with "CK1" replaced by the appropriate designator.
40
41
    /* CaveKey1 */
42
43
    #define ScemaKeyLengthCK1 8
    #define NeededLengthCK1 8
44
45
    #if NeededLengthCK1 > ScemaKeyLengthCK1
46
       #error NeededLengthCK1 too large
47
    #endif
48
49
    #if NeededLengthCK1 % 2
       #error NeededLengthCK1 must be an even number
51
    #endif
52
```

2.9.3. SCEMA Encryption/Decryption Procedure (Level 1)

2	Procedure name:			
3	SCEMA			
4	Inputs from calling process:			
5	msg_buf[n]	n*8 bits, n > 2		
6	csync[0-1]	32		
7	id	1 octet		
8	idMask	1 octet		
9	decrypt schedPtr	1 bit		
10	Scheden	pointer to key schedule containing scemaKey, obox,		
11 12		offKey, and neededLength		
12		officey, and needed engin		
13	Inputs from internal stored data	a:		
14	None.			
15	Outputs to calling process:			
16	msg_buf[n]	n*8 bits		
17	Outputs to internal stored data:	:		
18	None.			
19	This algorithm encrypts and	decrypts messages that are of length n		
20	octets, where $n > 2$.			
21		n an n-octet buffer called msg_buf[],		
22		d to one "msg_buf[]" value. The input		
23		variable csync should have a unique value for each message that is		
24		encrypted, with the portion that varies quickly in its lower 16 bits. The		
25	same value of csync is used a	same value of csync is used again for decryption.		
	The mercuraters id and ideal.	allow the internal course of the territory		
26	of cruptosyme to be forced to	c allow the internal copy of the top octet		
27		of cryptosync to be forced to a given value, idMask defines which bits		
28 29		are forced, and id defines the values of those bits. These inputs allow differentiation of scema instances. In particular, the following are		
30		differentiation of scema instances. In particular, the following are differentiated: instances within a single procedure, and those with		
31		different content, direction or architecture. By doing this, a class of		
32		attacks is prevented that use recurring encryptor/decryptor outputs. One		
33	well-known member of this cla			
34		the SCEMA variable-length session key		
35		ges via an enhanced CMEA algorithm.		
36	The process of SCMEA key ge	eneration is described in §2.9.1.		

decryption.

The decrypt variable shall be set to 0 for encryption, and to 1 for

SCEMA is given a pointer, schedPtr, to the desired key schedule structure. The structure contains the following elements: *scemaKey, 2 *obox, *offKey, and neededLength The first three are pointers to keys 3 (cryptovariables). The fourth, neededLength, generally corresponds to the true entropy of the key. A key generation mechanism may be implemented such that it outputs the scemaKey into a constant buffer size, independent of the true strength of the key. This parameter allows the stbox() function's iterations to track the true strength of the key, 8 which in turn allows for faster operation with lower strength keys. 9 The function stbox() is frequently used in SCEMA. For example, in 10 the case of an 8-octet SCEMA Key, stbox() is defined as: 11 12 k4)XOR k3)-k2)XOR k1)-k0 3 13 where "+" denotes modulo 256 addition, 14 "-" denotes modulo 256 subtraction, 15 "XOR" is the XOR function, 16 "z" is the function argument, 17 k0,...k7 are the eight octets of SCEMA key, 18 and I() is the outcome of the ibox 8-bit table look-up (see 19 Exhibit 2-54). 20

Exhibit 2-54 SCEMA with subtending functions stbox and SCEMA transform

```
/* SCEMA source including transforms and stbox "scema.c" */
3
    #include "cave.h" /* see Exhibit 2-2 */
5
    #include "scema.h" /* see Exhibit 2-53 */
6
    /* Stbox function
8
    Note: The SCEMA Key Length must be an even number of octets.
9
10
    The "-1" in the first "while" statement prevents overflow if
11
    ScemaKeyLength is accidentally odd.
12
13
    unsigned char stbox(const unsigned char z,
14
                     const unsigned char *scema_key,
15
                     const unsigned char len)
16
17
        unsigned char t = z;
18
        int i = 0;
19
20
        while(i < len - 1)</pre>
21
22
           t = ibox[(t + scema_key[i++]) & 0xff];
t = ibox[t ^ scema_key[i++]];
23
24
25
26
        --i;
27
28
        while(i > 1)
29
30
           t = ibox[(t - scema_key[--i]) & 0xff];
31
           t = ibox[t ^scema_key[--i]];
32
33
34
        t = (t - scema key[--i]) & 0xff;
35
        return t;
36
37
38
```

```
/* Transformation */
1
2
    void SCEMA transform(unsigned char *msg buf, const int octet count,
3
                      unsigned int *offInt, const unsigned char *key,
                      const unsigned int *obox, const unsigned char len)
6
       unsigned char k, z;
7
       int msg index;
8
9
       for (msg index = 0; msg index < octet count; msg index++)</pre>
10
11

angle \star offset generator cycle and involutary lookup of present octet \star /
12
13
          offsetInt[0] += offsetInt[1] + obox[offsetInt[1] & 0x0F];
14
          offsetInt[1] ^=
15
              ((offsetInt[0] \& 0xFFFF) >> 4) + (offsetInt[0] << 4);
16
17
          msg buf[msg index] = OFFSETB ^
18
              stbox((unsigned char)(msq buf[msq index] ^ OFFSETA),
19
                 key, len);
20
21
          /* bit-trade between present octet and the one below */
22
23
          if (msq index > 0)
24
25
              k = msg_buf[msg_index - 1] ^ msg_buf[msg_index];
26
              k &= stbox((unsigned char)(k ^ OFFSETA), key, len);
27
              msg buf[msg index - 1] ^= k;
28
              msg_buf[msg_index] ^= k;
29
30
31
           /* random octet permutation */
32
          /* exchange previous octet with a random one below it */
33
34
          if (msq index > 1)
35
36
              k = stbox((unsigned char) (msg buf[msg index] ^ OFFSETA),
37
                 key, len);
38
              k = ((msg_index) * k) >> 8;
39
              z = msg\_buf[k];
40
              msg buf[k] = msg_buf[msg_index - 1];
41
              msg buf [msg index - 1] = z;
42
           }
43
       }
44
45
       /* final octet permutation */
46
       /* exchange last octet with a random one below it */
47
48
       k = stbox((unsigned char)(0x37 ^ OFFSETA), key, len);
49
       k = ((msg index) * k) >> 8;
50
       z = msg_buf[k];
51
       msg buf[k] = msg buf[msg index - 1];
52
       msg buf [msg index - 1] = z;
53
54
```

```
/* final involution and XORing */
1
        k = stbox(msg_buf[0], key, len);
        for (msg_index = 1; msg_index < octet_count; msg_index++)</pre>
           msg_buf[msg_index] = stbox(msg_buf[msg_index], key, len);
6
           k = msg buf[msg index];
8
9
10
        msg_buf[0] = k;
        for (msg_index = 1; msg_index < octet_count; msg_index++)
   msg_buf[msg_index] ^= k ;</pre>
11
12
13
    }
14
15
```

```
/* Inverse Transformation */
1
2
    void SCEMA inv transform(unsigned char *msg buf,
3
                               const int octet count,
                               unsigned int *offInt,
6
                               const unsigned char *key,
                               const unsigned int *obox,
7
                               const unsigned char len)
8
9
       unsigned char k, z;
10
       int msg index;
11
12
       /* inverse of final involution and XORing */
13
14
       for (msg_index = 1; msg_index < octet_count; msg_index++)</pre>
15
          msg buf[msg index] ^= msg buf[0];
16
17
       for (msg index = 1; msg index < octet count; msg index++)</pre>
18
19
          msg buf[0] ^= msg buf[msg index];
20
          msg buf[msg index] = stbox(msg buf[msg index], key, len);
21
22
       msq buf[0] = stbox(msq buf[0], key, len);
23
24
       /* initial octet permutation */
25
       /* exchange last octet with a random one below it */
26
27
       k = stbox((unsigned char)(0x37 ^ OFFSETA), key, len);
28
       k = ((octet count) * k) >> 8;
29
       z = msg buf[k];
30
       msg buf[k] = msg buf[octet count - 1];
31
       msg buf[octet count - 1] = z;
32
33
       for (msg index = octet count - 1; msg index >= 0; msg index--)
34
35
           /* random octet permutation */
36
          /* exchange previous octet with a random one below it */
37
38
          if (msg index > 1)
39
40
              k = stbox((unsigned char) (msg buf[msg index] ^ OFFSETA),
41
                 key, len);
42
              k = ((msg index) * k) >> 8;
43
44
              z = msg buf[k];
              msg buf[k] = msg buf[msg index - 1];
45
              msq buf [msq index - 1] = z;
46
47
48
          /* bit-trade between present octet and the one below */
49
50
          if (msg index > 0)
51
52
              k = msg_buf[msg_index - 1] ^ msg_buf[msg_index];
53
              k &= stbox((unsigned char)(k ^ OFFSETA), key, len);
54
55
              msg buf [msg index - 1] ^= k;
             msg_buf[msg_index] ^= k;
56
57
58
```

```
/* involutary lookup of present octet and offset generator cycle */
1
          msq buf[msq index] = OFFSETA ^
              stbox((unsigned char)(msg buf[msg index] ^ OFFSETB),
                 key, len);
          offsetInt[1] ^=
              ((offsetInt[0] & 0xFFFF)>>4) + (offsetInt[0]<<4);
8
          offsetInt[0] -= offsetInt[1] + obox[offsetInt[1] & 0x0F];
9
       }
10
11
12
    /* SCEMA Algorithm */
13
14
    void SCEMA (unsigned char *msg buf,
15
                 const int octet count,
16
                 const unsigned char *csync,
17
                 const unsigned char id,
18
                 const unsigned char idMask,
19
                 const unsigned int decrypt,
20
                 keySched *schedPtr)
21
22
       unsigned char k, z, offsetc;
23
       int msq index;
24
       unsigned char *key;
25
       unsigned int *obox, *offKeyAux;
26
       unsigned char len;
27
       unsigned char csync3id;
28
       unsigned int csyncInt[2];
29
30
       /* load key schedule element pointers */
31
32
       key = schedPtr->scemaKey;
33
       obox = schedPtr->obox;
34
       offKeyAux = schedPtr->offKey;
35
       len = schedPtr->neededLength;
36
37
38
       /* Offset Generator Initialization */
39
40
       csync3id = (csync[3] & ~idMask) | (id & idMask);
41
42
43
       csyncInt[0] = (unsigned int)((csync[1] << 8) | (csync[0] & 0xFF));</pre>
       csyncInt[1] = (unsigned int)((csync3id << 8) | (csync[2] & 0xFF));</pre>
44
45
       offsetInt[0] = csyncInt[1] + offKeyAux[0];
46
       offsetInt[1] = csyncInt[0] + offKeyAux[1];
47
48
49
       offsetInt[0] += obox[offsetInt[1] & 0x0F]
                    + obox[(offsetInt[1] >> 4) & 0x0F]
50
                    + obox[(offsetInt[1] >> 8) & 0x0F]
51
                    + obox[(offsetInt[1] >> 12) & 0x0F] ;
52
53
       offsetInt[1] += obox[offsetInt[0] & 0x0F]
54
55
                    + obox[(offsetInt[0] >> 4) & 0x0F]
56
                    + obox[(offsetInt[0] >> 8) & 0x0F]
                    + obox[(offsetInt[0] >> 12) & 0x0F];
57
58
```

```
/* initial transformation */
1
       if (decrypt)
2
           SCEMA_transform(msg_buf, octet_count, offsetInt + 1, key,
3
              obox, len);
       else
5
           SCEMA transform (msq buf, octet count, offsetInt, key, obox, len);
6
7
8
       /* CMEA */
9
       offsetc = (unsigned char)((offsetInt[0] + offsetInt[1]) & 0xFF);
10
       /* first manipulation (inverse of third) */
11
       z = 0;
12
       for (msg index = 0; msg index < octet count; msg index++)</pre>
13
14
          k = stbox((unsigned char)(z ^ offsetc), key, len);
15
          msg buf[msg index] += k;
16
          z = msg buf[msg index];
17
18
19
       /* second manipulation (self-inverse) */
20
       for (msg_index = 0; msg_index < octet_count - 1; msg_index += 2)</pre>
21
          msg buf[msg index] ^= msg buf[msg index + 1];
22
23
       /* third manipulation (inverse of first) */
24
       z = 0;
25
       for (msg index = 0; msg index < octet count; msg index++)</pre>
26
27
          k = stbox((unsigned char)(z ^ offsetc), key, len);
28
          z = msg_buf[msg_index];
29
          msg buf[msg index] -= k;
30
31
32
       /* final inverse transformation */
33
       if (decrypt)
34
           SCEMA inv transform(msg buf, octet count, offsetInt, key,
35
              obox, Ten);
36
       else
37
          SCEMA inv transform(msg buf, octet count, offsetInt + 1, key,
38
              obox, len);
39
40
41
```

2.9.4. Block and KSG Encryption Primitives (Level 2)

These Level 2 primitives call SCEMA at Level 1 and are called by the voice privacy and message encryption procedures at Level 3.

2.9.4.1. SCEMA KSG

25

26

27

28

29

30

31

7	21514111 0021111/11100
5	Procedure name:
6	SCEMA_KSG
7	Inputs from calling process:
8	keystreamBuf[n] n octets, $1 \le n \le 256$
9	requestedStreamLen 1 - 256
10	inputBuf[n] 1 - 6 octets
11	inputLen 1 octet
12	contentType 1 octet defining voice or message
13	schedPtr pointer to SCEMA key schedule direction 1 bit
14	direction 1 bit
15	Inputs from internal stored data:
16	None.
17	Outputs to calling process:
18	keystreamBuf [n] n octets, 1 <= n <= 256
19	Outputs to internal stored data:
20	None.
21	This encryption primitive generates a buffer of keystream of length
22	requestedStreamLen based on the value of input buffer inputBuf[n] of
23	length inputLen. It runs SCEMA in a KSG mode where the input is fed
24	to both SCEMA's PT (plaintext) input and its CS (cryptosync) input.

The content type variable allows it to generate unique keystream depending upon whether it is used in voice privacy or message encryption. (This primitive is not called in RLP encryption (Enhanced Data Encryption).)

The pointer schedPtr is the SCEMA key schedule pointer described earlier in Section 2.9.

Direction indicates either the forward channel by 1, or the reverse channel by 0.

Exhibit 2-55 SCEMA KSG for Voice and Message Content

```
/* SCEMA KSG for Voice and Message Content "scemaKSG.c" */
2
3
    #include "scema.h" /* see Exhibit 2-53 */
4
5
    void SCEMA KSG(unsigned char *keystreamBuf,
6
                 const unsigned int requestedStreamLen,
7
8
                 const unsigned char *inputBuf,
9
                 const unsigned int inputLen,
10
                 const unsigned char contentType,
                 keySched *schedPtr,
11
                 const unsigned int direction)
12
13
       unsigned int i;
14
       unsigned char csync[4];
15
       unsigned char id;
16
       unsigned int outputStreamLen;
17
18
       /* Generates a minimum of 6 octets of keystream */
19
20
       outputStreamLen = MAX(SixOctets, requestedStreamLen);
21
       /* Combine ID segments */
22
       id = (unsigned char) (direction << 7) | contentType;</pre>
23
24
       /* Repeat input across SCEMA's PT field */
25
       for (i = 0; i < outputStreamLen; i++)</pre>
26
          keystreamBuf[i] = inputBuf[i % inputLen];
27
28
29
       Copy 4 least significant octets of PT to CS input.
30
31
       ID is XORed in to yield KSGs that are unique with
       respect to content and direction.
32
33
34
       for (i = 0; i < 4; i++)
          csync[i] = keystreamBuf[i] ^ id;
35
36
       SCEMA (keystreamBuf, outputStreamLen, csync, KSGArchitecture, KSGIdMask,
37
              ENCRYPTING.schedPtr);
38
    }
39
40
```

2.9.4.2. Long Block Encryptor

2	Procedure name:		
3	Long_Block_Encryptor		
4	Inputs from calling process:		
5	contentBuf[n]	6 octets	
6	contentType	1 octet defining voice or message	
7	decrypt	1 bit	
8	schedPtr	pointer to SCEMA key schedule	
9	direction	1 bit	
10	Inputs from internal stored data:		
11	None.		
12	Outputs to calling process:		
13	contentBuf [n]	6 octets	
14	Outputs to internal stored data:		
15	None.		
16		crypts or decrypts a 6-octet buffer by	
17		The content type variable allows it	
18		pending upon whether it is used in	
19	voice privacy or message encryption. (This primitive is not called in		
20	RLP encryption (Enhanced Data En	ncryption).)	
21	The parameter decrypt is set to 0 for	or encryption and 1 for decryption. It	
22	is needed here to determine the instance id number. This number		
23	uniquely identifies the particular SCEMA instance to prevent certain		
24	types of attacks.	•	
25		MA key schedule pointer described	
26	earlier in Section 2.9.		
27		ward channel by 1, or the reverse	
28	channel by 0.		

```
Exhibit 2-56 Long Block Encryptor for Voice and Message Content
2
3
    Long Block Encryptor (6 octets) for Voice and Message Content
    "longBlock.c"
    Note: The Long Block Encryptor/Decryptor's LHS and RHS are each 3 octets
6
    in length.
7
8
    #include "scema.h" /* see Exhibit 2-53 */
9
10
    void Long_Block_Encryptor(unsigned char *contentBuf,
11
                    const unsigned char contentType,
12
                    const unsigned int decrypt,
13
                    keySched *schedPtr,
14
                    const unsigned int direction)
15
16
       unsigned char csync[4] = \{0x00, 0x00, 0x00, 0x00\};
17
       unsigned char id;
18
       unsigned char instanceId;
19
20
21
    Combine ID segments
22
23
    Note: In particular, the LongBlockArchitecture ID segment forces bit 2
    of SCEMA's cryptosync top octet to 1 to differentiate it from all
24
    other uses (i.e.KSG uses) where bit 2 is forced to 0.
25
26
       id = (unsigned char) (direction << 7) | contentType |</pre>
27
          LongBlockArchitecture;
28
29
30
    SCEMA instance 0: PT <- LHS of contentBuf, CS <- RHS, instance = 0
31
    for encrypt, and 2 for decrypt.
32
33
    Note: The temporary variable csync is used to prevent buffer overflow
34
    during reading since SCEMA reads in a 4-octet csync buffer. This is
35
    not needed in the second instance since no overflow occurs and since
36
    the highest cync input octet is zeroed by LongBlkIdMask.
37
    * /
38
39
       csync[0] = contentBuf[3];
40
       csync[1] = contentBuf[4];
41
       csync[2] = contentBuf[5];
42
43
       if (decrypt)
44
          instanceId = id | Instance2;
45
       else
46
          instanceId = id;
47
48
       SCEMA(contentBuf, ThreeOctets, csync, instanceId, LongBlkIdMask,
49
              decrypt,schedPtr);
50
51
    /* SCEMA instance 1: PT <- RHS of contentBuf, CS <- LHS, instance = 1 */
52
53
       instanceId = id | Instance1;
54
55
       SCEMA(contentBuf + 3, ThreeOctets, contentBuf, instanceId,
56
          LongBlkIdMask, decrypt, schedPtr);
57
```

```
/* SCEMA instance 2: PT <- LHS of contentBuf, CS <- RHS, instance = 2 */
1
2
       csync[0] = contentBuf[3];
3
       csync[1] = contentBuf[4];
       csync[2] = contentBuf[5];
       if (decrypt)
          instanceId = id;
8
       else
9
10
          instanceId = id | Instance2;
11
       SCEMA(contentBuf, ThreeOctets, csync, instanceId, LongBlkIdMask,
12
             decrypt,schedPtr);
13
14
    }
15
16
```

2.9.4.3. Short Block Encryptor

2	Procedure name:			
3	Short_Block_Encryptor			
4	Inputs from calling process:			
5	contentBuf[n]	1 - 6 octets, $1 - 47$ bits		
6	numBits	1 - 47 number of content bits in		
7		contentBuf buffer		
8	contentType	1 octet defining voice or message		
9	entropy[4] decrypt	4 octets of possible added entropy 1 bit		
10 11	schedPtr	pointer to SCEMA key schedule		
12	direction	1 bit		
13	Inputs from internal stored dat	a:		
14	None.			
15	Outputs to calling process:			
16	contentBuf [n]	1 - 6 octets, 1 – 47 bits		
17	Outputs to internal stored data	:		
18	None.			
19		ock encrypts or decrypts a 1- to 6 octet		
20		um of 1 bit and a maximum of 47 bits.		
21	•	t the Short Block Encryptor will never be		
22 23	48 bits.)	e the Long Block Encryptor is used for		
	The Chart Dieds energyter on	d door not or or formed from four Foistal		
24 25		d decryptor are formed from four Feistel XSG mode. The Feistel piece contains the		
26 26		the input buffer, output buffer, a KSG		
27		s, an instance ID used for differentiating		
28	SCEMA uses according to in	nstances, direction, and content, entropy		
29		from message type and RAND if extant for the type of content being		
30	encrypted, and a pointer to the	key schedule.		
31	The contentType parameter	allows the Short Block Encryptor to		
32	generate unique keystream de	pending upon whether it is used in voice		
33		on. (This primitive is not called in RLP		
34	encryption (Enhanced Data En	ecryption).)		
35		ed in for message encryption where the		
36		RAND (for DCCH only) provide added		
37	entropy to the encryption.			

The parameter decrypt is set to 0 for encryption and 1 for decryption. It is needed here to determine the instance id number. This number

2

3

4

5

6

7

8

10

uniquely identifies the particular SCEMA instance to prevent certain types of attacks. Also, the encryptor and decryptor architectures are not isomorphic due to the four instances of SCEMA (Feistel pieces), and thus the decryptor parameter is needed to select the architecture.

The pointer schedPtr is the SCEMA key schedule pointer described earlier in Section 2.9.

The direction parameter indicates either the forward channel by 1, or the reverse channel by 0.

Exhibit 2-57 Short Block Encryptor for Voice and Message Content

```
11
    Short Block Encryptor (less than 6 octets) for Voice and Message Content
12
    "shortBlock.c"
13
    Note: The Short Block Encryptor/Decryptor's LHS and RHS are each less
    then or equal 3 octets in length. The number of content-bearing bits of
16
    its LHS (left hand side) always equals or is one greater than the number
17
18
    of content-bearing bits in its RHS.
19
20
    #include "scema.h" /* see Exhibit 2-53 */
21
22
    void feistelPiece(const unsigned char *inputBuf,
23
24
                   unsigned char *outputBuf,
                   const unsigned char *ksgTemplate,
25
                   const unsigned char instanceId,
26
27
                   const unsigned char *entropy,
                   keySched *schedPtr)
28
29
       unsigned int i;
30
31
       unsigned char csync[4];
       unsigned char keystreamBuf[3];
32
33
34
35
       SCEMA's PT input is tied to CS input with ID differentiator..
       ID is XORed in to yield KSGs that are unique with respect
36
       to content, direction, and instance.
37
       */
38
39
40
       for (i = 0; i < 3; i++)
41
          csync[i] = inputBuf[i] ^ entropy[i];
42
          keystreamBuf[i] = csync[i] ^ instanceId;
43
44
45
       csync[3] = entropy[3] ^ instanceId;
46
47
       SCEMA (keystreamBuf, ThreeOctets, csync, KSGArchitecture, KSGIdMask,
48
          ENCRYPTING, schedPtr);
49
```

```
/* KSG output is XORed with right buffer. The template passes
1
       only those bits that correspond to the right buffer's content
2
       bits.
3
       */
       for (i = 0; i < 3; i++)
           outputBuf[i] ^= keystreamBuf[i] & ksqTemplate[i];
8
    }
9
10
11
    void Short Block Encryptor(unsigned char *contentBuf,
12
                    const unsigned int numBits,
13
                    const unsigned char contentType,
14
                    const unsigned char *entropy,
15
                    const unsigned int decrypt,
16
                    keySched *schedPtr,
17
                    const unsigned int direction)
18
19
       unsigned int i;
20
       unsigned char id;
21
       unsigned int numBitsLocal;
22
       unsigned int octetSize;
23
       unsigned int numTopBits;
24
25
       unsigned char leftBuf[3] = \{0x00,0x00,0x00\};
26
       unsigned char rightBuf[3] = \{0x00,0x00,0x00\};
27
       unsigned int leftBufNumBits;
28
       unsigned int rightBufNumBits;
29
30
       unsigned char leftKsgTemplate[3] = \{0x00,0x00,0x00\};
31
       unsigned char rightKsgTemplate[3] = \{0x00,0x00,0x00\};
32
33
       unsigned char *pContent;
34
       unsigned char *pLeft;
35
       unsigned char *pRight;
36
37
       /* Prevents accidental buffer overflow */
38
39
       numBitsLocal = MIN(numBits,48);
40
       numBitsLocal = MAX(numBitsLocal,1);
41
42
43
       Number of octets needed to contain contentBuf bits
44
       Note: The index of the top octet (the highest one containing
45
       content) is thus octetSize - 1.
46
47
48
       octetSize = ((numBitsLocal - 1) / 8) + 1;
49
50
51
       Number of content bits in top octet which occupy the top
52
       bits of the octet
53
       * /
54
55
       numTopBits = numBitsLocal - (8 * (octetSize - 1));
56
57
       /* Number of content bits in left buffer */
58
59
       leftBufNumBits = (numBitsLocal + 1)/2;
60
61
```

```
/* Number of content bits in right buffer */
1
       rightBufNumBits = numBitsLocal/2;
       /* Ensure that unused contentBuf octets are zeroed and that
       unused bits in the top octet are zeroed.
6
8
       for (i = octetSize; i < 6; i++)
9
          contentBuf[i] = 0;
10
11
       contentBuf[octetSize - 1] >>= (8 - numTopBits);
12
       contentBuf[octetSize - 1] <<= (8 - numTopBits);</pre>
13
14
15
       Divide contentBuf input bits between left and right buffers
16
       to begin building a Feistel network. If numBitsLocal is even,
17
       both buffers receive an equal number of bits. If numBitsLocal
18
       is odd, the left buffer receives one more bit than the right
19
       buffer.
20
       * /
21
22
       pContent = contentBuf;
23
       pLeft = &leftBuf[0];
24
25
       pRight = &rightBuf[0];
26
27
       for (i = 0; i < 3; i++)
28
           *pLeft |= *pContent & 0xAA;
29
           *pRight |= (*pContent++ & 0x55) << 1;
30
           *pLeft++ \mid = (*pContent & 0xAA) >> 1;
31
           *pRight++ \mid = *pContent++ & 0x55;
32
33
34
       /* Now that the content has been extracted from the contentBuf,
35
       the buffer is re-used temporarily to generate KSG templates.
36
       These templates will be used to pass only those KSG bits
37
       corresponding to the content-bearing left and right buffer bits.
38
39
40
       for (i = 0; i < octetSize; i++)
41
           contentBuf[i] = 0xFF;
42
43
       for (i = octetSize; i < 6; i++)
44
           contentBuf[i] = 0;
45
46
       contentBuf[octetSize - 1] >>= (8 - numTopBits);
47
       contentBuf[octetSize - 1] <<= (8 - numTopBits);</pre>
48
49
       pContent = contentBuf;
50
       pLeft = &leftKsqTemplate[0];
51
       pRight = &rightKsgTemplate[0];
52
53
```

```
for (i = 0; i < 3; i++)
1
2
           *pLeft |= *pContent & 0xAA;
3
           *pRight |= (*pContent++ & 0x55) << 1;
           *pLeft++ |= (*pContent & 0xAA) >> 1;
5
           *pRight++ \mid = *pContent++ & 0x55;
6
        }
7
8
        /*
9
        Combine ID segments. A DCCH/DTC id segment is not needed for
10
        differentiation because the two channels use different keys.
11
        */
12
13
        id = (unsigned char)(direction << 7) | contentType;</pre>
14
15
16
        /*
17
        Encryption/Decryption
18
19
        * /
20
21
        if(!decrypt) /* encrypting */
22
23
24
           Four Feistel-SCEMA instances. The zeroth instance does not
25
           contain an explicit instance number because the number
26
27
           is zero.
           */
28
29
           feistelPiece(leftBuf, rightBuf, rightKsgTemplate,
30
31
                        id, entropy, schedPtr);
32
           feistelPiece(rightBuf,leftBuf,leftKsgTemplate,
33
                         (unsigned char) (id | Instance1),
34
                        entropy,schedPtr);
35
36
           feistelPiece(leftBuf,rightBuf,rightKsgTemplate,
37
                         (unsigned char) (id | Instance2),
38
                        entropy,schedPtr);
39
40
           feistelPiece(rightBuf,leftBuf,leftKsgTemplate,
41
                        (unsigned char) (id | Instance3),
42
43
                        entropy, schedPtr);
44
        }
45
46
47
48
        Almost everything above is done in reverse order.
49
50
51
```

```
else /* decrypting */
1
2
           feistelPiece(rightBuf,leftBuf,leftKsgTemplate,
3
                        (unsigned char) (id | Instance3),
                        entropy,schedPtr);
6
           feistelPiece(leftBuf,rightBuf,rightKsgTemplate,
                        (unsigned char) (id | Instance2),
8
                        entropy,schedPtr);
9
10
           feistelPiece(rightBuf,leftBuf,leftKsgTemplate,
11
                        (unsigned char) (id | Instance1),
12
                        entropy,schedPtr);
13
14
           feistelPiece(leftBuf,rightBuf,rightKsgTemplate,
15
                        id, entropy, schedPtr);
16
17
       }
18
19
20
       Output processing: Load left and right buffers back into content
21
22
       buffer.
        */
23
24
       for (i = 0; i < 6; i++)
25
26
           contentBuf[i] = 0;
27
       pContent = contentBuf;
28
       pLeft = &leftBuf[0];
29
       pRight = &rightBuf[0];
30
31
       for (i = 0; i < 3; i++)
32
33
           *pContent |= *pLeft & 0xAA;
34
           *pContent++ |= (*pRight >> 1) & 0x55;
35
           *pContent |= (*pLeft++ << 1) & 0xAA;
36
           *pContent++ |= *pRight++ & 0x55;
37
38
39
40
41
```

2.9.5. Voice, Message, and Data Encryption Procedures (Level 3)

These top-level procedures interface directly TIA/EIA-136-510 and call the Level 2 procedures and, in the case of Enhanced Data Encryption only, the Level 1 (SCEMA) procedure.

2.9.5.1. Enhanced Voice Privacy

Procedure name:	
Enhanced_Voice_Privacy	
Inputs from calling process:	
coderVer speechBuf1[n] num1aBits speechBufRem [n] numRemBits decrypt keyGenerator direction	0, 1, 2, etc. n octets, 1 <= n <= 256 n >= 1 n octets, 0 <= n <= 256 n >= 0 1 bit 1,2,3, etc. 1 bit
Inputs from internal stored data:	
None.	
Outputs to calling process:	
speechBuf1[n] speechBufRem [n]	n octets, 1 <= n <= 256 n octets, 0 <= n <= 256
Outputs to internal stored data:	
None.	

This Level 3 procedure encrypts or decrypts a frame of speech. The frame is separated into two buffers, speechBufl and speechBufRem, containing speech coders' Class 1A and remaining (Class 1B and 2) bits, respectively. Class 1A bits are those that are protected by a CRC in the speech coder algorithm. The respective numbers of these bits are num1aBits and numRemBits.

The parameter coderVer is set to 0 in TIA/EIA-136-510 and is not used here. It comprises a hook in case the CCA would ever need to be revised in the future due to a speech coder architecture incompatible with this current procedure.

The parameter decrypt is set to 0 for encryption and 1 for decryption. The encryptor and decryptor architectures are not isomorphic and thus the decryptor parameter is needed to select the architecture.

2

3

4

5

6

8

10

11

12 13

14

15

16

The parameter keyGenerator is currently set to 1 in TIA/EIA-136-510 to indicate CaveKey1, a key schedule based on the current CAVE algorithm running at its full strength. Internal to this procedure, the parameter is used to point to the DTCKey CaveKey1.

Direction indicates either the forward channel by 1, or the reverse channel by 0.

If the number of Class 1A bits is 48, then this procedure calls the Long Block Encryptor for these bits. If the number is greater than 48, the excess above 48 are encrypted by the SCEMA KSG. However, prior to encryption, their entropy is folded in to the first 48 bits that are encrypted by the Long Block Encryptor.

If the number of Class 1A bits is less than 48, these bits are encrypted by the Short Block Encryptor.

The remaining bits are encrypted by the SCEMA KSG using the Class 1A ciphertext as input (entropy).

Exhibit 2-58 Enhanced Voice Privacy

```
/* Enhanced Voice Privacy "enhVoicePriv.c" */
17
18
    #include "scema.h" /* see Exhibit 2-53 */
19
20
    void Enhanced Voice Privacy(const unsigned int coderVer,
21
22
                          unsigned char *speechBuf1,
                           const unsigned int numlaBits,
23
                          unsigned char *speechBufRem,
24
                           const unsigned int numRemBits,
25
                           const unsigned int decrypt,
26
                           const unsigned int keyGenerator,
27
                          const unsigned int direction)
28
29
       unsigned int i;
30
       unsigned char keystreamBuf[MaxFrameOctetSize];
31
       unsigned int net1aOctetSize;
32
       unsigned int numlaTopBits;
33
       unsigned int excess1aOctetSize;
34
       unsigned int remBitsOctetSize;
35
       unsigned int numRemTopBits;
36
       unsigned int ksgInputOctetSize;
37
       unsigned char nullEntropy[4] = \{0x00, 0x00, 0x00, 0x00\};
38
39
       /* Pointers to be set and used later */
40
41
       unsigned char *pKeyStream;
42
       unsigned char *pSpeech;
43
44
45
       Number of octets that contain the Class 1A bits, and
46
       number of bits in the 1A bits top octet.
47
48
49
       netlaOctetSize = ((numlaBits - 1) / 8) + 1;
50
       numlaTopBits = numlaBits - (8 * (netlaOctetSize - 1));
```

Information disclosed in this document is subject to the export jurisdiction of the US Department of Commerce as specified in Export Administration Regulations (title 15 CFR parts 730 through 774 inclusive).

```
1
2
       Number of octets that contain any excess Class 1A bits
3
       beyond the first 6 octets (48 bits). For ACELP and VSELP,
       this equals zero.
6
       * /
       excess1aOctetSize = MAX(net1aOctetSize ,6) - 6;
8
9
10
       Number of octets that contain the remaining bits, namely
11
       those bits not protected by a CRC, usually called Class 1B
12
       and Class 2 bits. Also calculated is the number of bits
13
       in the remaining bits top octet.
14
15
16
       remBitsOctetSize = ((numRemBits - 1) / 8) + 1;
17
       numRemTopBits = numRemBits - (8 * (remBitsOctetSize - 1));
18
19
20
       If the number of Class 1A bits is greater than or equal
21
       to 48 bits, the 6-octet Long Block Encryptor is used, and
22
       its output feeds the KSG. However, if the number of 1A bits
23
       is less than 48 bits, the Short Block Encryptor is used and
24
       only its output is fed to the KSG. In this latter case, the
25
26
       KSG input will be repeated as necessary (in SCEMA KSG()) to
       fill SCEMA's plaintext input field.
27
       */
28
29
       ksgInputOctetSize = MIN(net1aOctetSize, 6);
30
31
       /* Input clean up */
32
33
34
       Ensure that bits other than the content-containing
35
       1A top bits are zeroed.
36
37
38
       speechBuf1[net1aOctetSize - 1] >>= (8 - numlaTopBits);
39
       speechBuf1[net1aOctetSize - 1] <<= (8 - num1aTopBits);</pre>
40
41
       /*Do the same for the remaining bits, i.e the Class 1B and
42
43
       Class 2 bits.
       * /
44
45
       speechBufRem[remBitsOctetSize - 1] >>= (8 - numRemTopBits);
46
       speechBufRem[remBitsOctetSize - 1] <<= (8 - numRemTopBits);</pre>
47
48
```

```
if(!decrypt) /* encrypting */
1
2
3
          If there are more than 48 1A bits, XOR the excess
          into initial 48 bits to inject added entropy.
6
          for (i = 0; i < excess1aOctetSize; i++)</pre>
8
              speechBuf1[i % 6] ^= speechBuf1[i + 6];
9
10
11
          Use different block encryptors depending on the number
12
          of 1A bits.
13
           * /
14
           if(num1aBits >= 48)
15
           {
16
17
              Block encrypt the first 6 octets of speechBuf1.
18
              Note: keyGenerator = 1 for CaveKey1. The first
19
              6 octets of speechBufl are replaced by ciphertext.
20
21
22
              Long Block Encryptor(speechBuf1, VoiceContent, decrypt,
23
                                 dtcScheds + keyGenerator - 1,
24
                                 direction);
25
           }
26
27
           else /* num1aBits < 48 */
28
29
30
              Block encrypt numlaBits of speechBufl to yield the
31
              same amount of ciphertext.
32
33
34
              Short Block Encryptor (speechBuf1, num1aBits, VoiceContent,
35
                                        nullEntropy, decrypt,
36
                                        dtcScheds + keyGenerator - 1,
37
38
                                        direction);
           }
39
40
41
          Form the appropriate amount of keystream with
42
          speechBufl as input. Either the first 6 octets
43
44
          of speechBufl are used which comprise the output of the
          Long Block Encryptor, or less are used if
45
          ksqInputOctetSize is set less than 6 octets, namely the
46
          output of the Short Block Encryptor.
47
           */
48
49
          SCEMA KSG(keystreamBuf,
50
                    excess1aOctetSize + remBitsOctetSize,
51
                    speechBuf1,ksgInputOctetSize,VoiceContent,
52
                    dtcScheds + keyGenerator - 1,direction);
53
54
55
          XOR keystream into buffers to yield ciphertext
56
          Start at zeroth keystream octet
57
           */
58
59
          pKeyStream = &keystreamBuf[0];
60
61
```

```
/* First encrypt excess 1A bits if extant */
1
           pSpeech = speechBuf1 + 6;
           for (i = 0; i < excess1aOctetSize; i++)</pre>
              *pSpeech++ ^= *pKeyStream++;
6
8
           Ensure that bits other than the content-containing
9
           (encrypted) (excess) 1A top bits are zeroed.
10
11
12
           speechBuf1[net1aOctetSize - 1] >>= (8 - numlaTopBits);
13
           speechBuf1[net1aOctetSize - 1] <<= (8 - num1aTopBits);</pre>
14
15
16
           /* Then encrypt remaining bits */
17
18
           pSpeech = speechBufRem;
19
20
           for (i = 0; i < remBitsOctetSize; i++)</pre>
21
              *pSpeech++ ^= *pKeyStream++;
22
23
        }
24
25
26
       else /* decrypting */
27
28
29
           Almost everything above is done in reverse order.
30
           The KSG is now first, and the block encryptor second.
31
           */
32
33
           SCEMA KSG(keystreamBuf,
34
                     excess1aOctetSize + remBitsOctetSize,
35
                     speechBuf1,ksgInputOctetSize,VoiceContent,
36
                     dtcScheds + keyGenerator - 1,direction);
37
38
           pKeyStream = &keystreamBuf[0];
39
          pSpeech = speechBuf1 + 6;
40
41
           for (i = 0; i < excess1aOctetSize; i++)</pre>
42
43
              *pSpeech++ ^= *pKeyStream++;
44
           pSpeech = speechBufRem;
45
46
           /* Decrypt remaining bits */
47
48
           for (i = 0; i < remBitsOctetSize; i++)</pre>
49
              *pSpeech++ ^= *pKeyStream++;
50
51
           /* Block encryptor choice */
52
53
           if(num1aBits >= 48)
54
55
              Long Block Encryptor(speechBuf1, VoiceContent, decrypt,
56
                                  dtcScheds + keyGenerator - 1,
57
                                  direction);
58
           }
59
60
```

```
else /* num1aBits < 48 */
1
              Short_Block_Encryptor(speechBuf1,num1aBits,VoiceContent,
3
                                     nullEntropy,decrypt,
                                     dtcScheds + keyGenerator - 1,direction);
           }
6
8
           Ensure that bits other than the content-containing
9
           (decrypted) 1A top bits are zeroed, and then do
10
           post-XORing.
11
           */
12
13
           speechBuf1[net1aOctetSize - 1] >>= (8 - num1aTopBits);
14
           speechBuf1[net1aOctetSize - 1] <<= (8 - num1aTopBits);</pre>
15
16
           if(num1aBits > 48)
17
              for (i = 0; i < excess1aOctetSize; i++)</pre>
18
                 speechBuf1[i % 6] ^= speechBuf1[i + 6];
19
20
        }
21
22
23
       Remaining output clean up: Ensure that bits other than the
24
       content-containing remaining bits (Class 1B and Class 2
25
26
       bits) are zeroed.
27
28
       speechBufRem[remBitsOctetSize - 1] >>= (8 - numRemTopBits);
29
       speechBufRem[remBitsOctetSize - 1] <<= (8 - numRemTopBits);</pre>
30
31
    }
32
33
```

27

28

29

31

32

33

35

36

37

2.9.5.2. Enhanced Message Encryption

			<u>-</u>	
2		Procedure name:		
3		Enhanced_Message_Encryption		
4]	Inputs from calling process:		
5		msgBuf [n]	n octets, 1 <= n <= 256	
6		numBits	n >= 1	
7		dechDTC	1 bit	
8		rand[4]	4 octets	
9		msgType	1 octet	
10		decrypt	1 bit	
11		keyGenerator	1,2,3, etc.	
12		direction	1 bit	
13		Inputs from internal stored data:		
14		None.		
15	ı	Outputs to calling process:		
16		msgBuf[n]	n octets, 1 <= n <= 256	
17		Outputs to internal stored data:		
18		None.		
19			or decrypts the Layer 3 content of a	
20			and its number of bits are denoted by	
21	1	the parameters msgBuf and numBi	ts respectively.	
22	,	The parameter dechDTC indicates	s to this procedure whether messages	
23			aDTC = 0), or on the DTC channel	
23			ption only, the value rand is used for	
			sgType (Message Type). For DTC	
25	•	auucu eniiopy iii auuiiion to m	sgrype (message rype). For DIC	

encryption, only msgType is used.

The parameter decrypt is set to 0 for encryption and 1 for decryption. The encryptor and decryptor architectures are not isomorphic and thus the decryptor parameter is needed to select the architecture.

The parameter keyGenerator is currently set to 1 in TIA/EIA-136-510 to indicate CaveKey1, a key schedule based on the current CAVE algorithm running at its full strength. Internal to this procedure, the parameter is used to point to the DTC CaveKeyl key schedule (DTCKey) for DTC messages, and to the DCCH CaveKey1 key schedule (DCCHKey) for DCCH messages.

Direction indicates either the forward channel by 1, or the reverse channel by 0.

If the number of message bits is 48, then this procedure calls the Long Block Encryptor for these bits. If this number is greater than 48, the excess above 48 are encrypted by the SCEMA KSG. However, prior to encryption, their entropy is folded in to the first 48 bits that are encrypted by the Long Block Encryptor.

If the number of message bits is less than 48, these bits are encrypted by the Short Block Encryptor.

Exhibit 2-59 Enhanced Message Encryption

```
9
    /* Enhanced Message Encryption "enhMsgEnc.c" */
10
    #include "scema.h" /* see Exhibit 2-53 */
11
12
    void Enhanced Message Encryption (unsigned char *msgBuf,
13
                           const unsigned int numBits,
14
15
                           const unsigned int dcchDTC,
                           const unsigned char *rand,
16
17
                           const unsigned char msgType,
                           const unsigned int decrypt,
18
19
                           const unsigned int keyGenerator,
                           const unsigned int direction)
20
21
       unsigned int i;
22
       unsigned char keystreamBuf[MaxMessageOctetSize];
23
       unsigned int msgBufOctetSize;
24
25
       unsigned int numTopBits;
26
       unsigned int excessOctetSize;
27
       unsigned int ksgInputOctetSize;
       unsigned char entropy[4] = \{0x00, 0x00, 0x00, 0x00\};
28
29
30
        /* Pointers to be set and used later */
31
       unsigned char *pKeyStream;
32
       unsigned char *pMessage;
33
       keySched *pDcchDtc;
34
35
        /* Entropy gathering and key schedule selection*/
36
37
        if(dcchDTC) /* DTC channel */
38
39
           entropy[0] = msgType;
40
           pDcchDtc = dtcScheds;
41
42
43
        else /* DCCH channel */
44
45
           for (i = 0; i < 4; i++)
46
           entropy[i] = rand[i];
entropy[0] ^= msgType;
47
48
           pDcchDtc = dcchScheds;
49
50
51
```

1

2

3

4

5

6

```
/*
1
       Number of octets that contain the message bits, and
2
       number of bits in their top octet.
3
       msqBufOctetSize = ((numBits - 1) / 8) + 1;
6
       numTopBits = numBits - (8 * (msgBufOctetSize - 1));
7
8
9
       Number of octets that contain any excess message bits
10
       beyond the first 6 octets (48 bits).
11
12
13
       excessOctetSize = MAX(msgBufOctetSize ,6) - 6;
14
15
16
       If the number of message bits is greater than or equal
17
       to 48 bits, the 6-octet Long Block Encryptor is used, and
18
       its output feeds the KSG. The KSG is run only if excess
19
       bits are present. However, if the number of message bits
20
       is less than 48 bits, only the Short Block Encryptor is
21
22
       used.
       */
23
24
25
       ksgInputOctetSize = MIN(msgBufOctetSize, 6);
26
27
       /* Input clean up */
28
29
       Ensure that bits other than the content-containing
30
31
       top bits are zeroed.
        * /
32
33
       msgBuf[msgBufOctetSize - 1] >>= (8 - numTopBits);
34
       msqBuf[msqBufOctetSize - 1] <<= (8 - numTopBits);</pre>
35
36
       if(!decrypt) /* encrypting */
37
38
       {
39
          If there are more than 48 message bits, XOR the excess
40
          into initial 48 bits to inject added entropy.
41
           * /
42
43
44
          for (i = 0; i < excessOctetSize; i++)
              msqBuf[i % 6] ^= msqBuf[i + 6];
45
46
47
          Use different block encryptors depending on the number
48
49
          of message bits.
           */
50
51
          if(numBits >= 48)
52
53
54
              Block encrypt the first 6 octets of msgBuf and
55
              first inject entropy.
56
              Note: keyGenerator = 1 for CaveKey1. The first
57
              6 octets of msgBuf are replaced by ciphertext.
58
              */
59
60
```

```
for (i = 0; i < 4; i++)
1
                  msgBuf[i] ^= entropy[i];
              Long Block Encryptor (msgBuf, MessageContent, decrypt,
                                   pDcchDtc + keyGenerator - 1,
6
                                   direction);
               if(numBits > 48)
8
9
10
                  Form the appropriate amount of keystream with
11
                  msgBuf as input.
12
13
14
                  SCEMA KSG(keystreamBuf, excessOctetSize, msgBuf,
15
                     ksgInputOctetSize, MessageContent,
16
                     pDcchDtc + keyGenerator - 1, direction);
17
18
19
                  XOR keystream into buffers to yield ciphertext
20
                  Start at zeroth keystream octet
21
22
23
                  pKeyStream = &keystreamBuf[0];
24
25
                  /* First encrypt excess message bits if extant */
26
27
                  pMessage = msgBuf + 6;
28
                  for (i = 0; i < excessOctetSize; i++)
  *pMessage++ ^= *pKeyStream++;</pre>
29
30
31
           }
32
33
           else /* numBits < 48 */
34
35
36
              Block encrypt numBits of msgBuf to yield the
37
              same amount of ciphertext.
38
39
40
              Short_Block_Encryptor(msgBuf, numBits, MessageContent,
41
                  entropy, decrypt, pDcchDtc + keyGenerator - 1,direction);
42
43
           }
44
           /*
45
           Ensure that bits other than the content-containing
46
           (encrypted) (excess) message top bits are zeroed.
47
           */
48
49
           msgBuf[msgBufOctetSize - 1] >>= (8 - numTopBits);
50
           msgBuf[msgBufOctetSize - 1] <<= (8 - numTopBits);</pre>
51
52
        }
53
54
```

```
else /* decrypting */
1
2
3
           Almost everything above is done in reverse order.
           The KSG is now first, and the block encryptor second.
5
6
           * /
           if(numBits > 48)
8
9
10
11
       SCEMA KSG(keystreamBuf,excessOctetSize,msgBuf,ksgInputOctetSize,
12
                 MessageContent, pDcchDtc + keyGenerator - 1, direction);
13
14
                 pKeyStream = &keystreamBuf[0];
15
                 pMessage = msgBuf + 6;
16
17
                 for (i = 0; i < excessOctetSize; i++)</pre>
18
                     *pMessage++ ^= *pKeyStream++;
19
20
21
22
           /* Block encryptor choice */
23
24
           if(numBits >= 48)
25
26
              Long_Block_Encryptor(msgBuf,MessageContent,decrypt,
27
                                  pDcchDtc + keyGenerator - 1,
28
                                  direction);
29
30
              for (i = 0; i < 4; i++)
31
                 msgBuf[i] ^= entropy[i];
32
           }
33
34
           else /* numBits < 48 */
35
36
              Short Block Encryptor (msqBuf, numBits, MessageContent, entropy,
37
38
                            decrypt, pDcchDtc + keyGenerator - 1,direction);
           }
39
40
41
           Ensure that bits other than the content-containing
42
           (decrypted) message top bits are zeroed, and then do
43
44
           post-XORing.
45
46
           msgBuf[msgBufOctetSize - 1] >>= (8 - numTopBits);
47
           msqBuf[msqBufOctetSize - 1] <<= (8 - numTopBits);</pre>
48
49
           if(numBits > 48)
50
              for (i = 0; i < excessOctetSize; i++)</pre>
51
                 msgBuf[i % 6] ^= msgBuf[i + 6];
52
53
        }
54
55
56
57
```

2.9.5.3. Enhanced Wireless Data Encryption

2	Procedure name:	
3	Enhanced_Data_Mask	
4	Inputs from calling process:	
5	mask[len]	len octets
6	HOOK	32 bits
7	len	1 <= len <= 256
8	keyGenerator	1,2,3, etc.
9	Inputs from internal stored da	ata:
10	None.	
11	Outputs to calling process:	
12	mask[len]	len octets
13	Outputs to internal stored dat	ta:
14	None.	
	Enhanced data are marken Co	. 126
15		r 136 wireless data services is provided by
16		acrypt mode as a KSG. This procedure
17		sk of length len octets, between 1 and 256
18	keystream mask of length len	e output value "mask" buffer containing
19	keysueam mask of length len	i ociets.
20	HOOK is a 32-bit value that	serves as cryptosync, and is input both to

21

22

23

24

25

26

27

28

SCEMA's cryptosync input and repeated across its plaintext field.

The parameter keyGenerator is currently set to 1 in TIA/EIA-136-510 to indicate CaveKey1, a key schedule based on the current CAVE algorithm running at its full strength. Internal to this procedure, the parameter is used to point to the DTC CaveKey1.

Internal to this procedure is a mechanism for differentiating this keystream from that produced by other uses of SCEMA in the KSG mode. To accomplish, it uses the identifier RlpContent.

Exhibit 2-60 Enhanced Data Mask Generation

```
/* Enhanced Data Mask Generation "enhDataMask.c" */
2
3
    #include "scema.h" /* see Exhibit 2-53 */
4
5
    void Enhanced Data Mask(unsigned char *mask,
6
                       const unsigned long HOOK,
7
8
                       const unsigned int len,
9
                       const unsigned int keyGenerator)
10
11
       unsigned int i;
       unsigned char csync[4];
12
       unsigned char maskSix[6];
13
14
       csync[0] = (unsigned char) (HOOK & 0xFF);
15
       csync[1] = (unsigned char)((HOOK >> 8) & 0xFF);
16
       csync[2] = (unsigned char)((HOOK >> 16) & 0xFF);
17
       csync[3] = (unsigned char)((HOOK >> 24) & 0xFF);
18
19
20
       if(len >= 6)
21
22
           /* Repeat HOOK across SCEMA's PT field */
23
          for (i = 0; i < len; i++)
24
              mask[i] = csync[i % 4];
25
26
           /* Prevents cross-replay effects with other content types */
27
          for (i = 0; i < 4; i++)
28
              csync[i] ^= RlpContent;
29
30
31
32
          Note: keyGenerator = 1 for CaveKey1.
          Since RLP encryption uses SCEMA in a KSG mode, the values
33
          KSGArchitecture and KSGIdMask are passed. This serves to force
34
35
          bit 2
                    in the cryptosync's top octet to zero to differentiate
          the cryptosync from that used in the Long Block Encryptor.
36
37
          SCEMA (mask, len, csync, KSGArchitecture, KSGIdMask, ENCRYPTING,
38
                 dtcScheds + keyGenerator - 1);
39
40
       }
41
42
43
44
    If requested length is less then 6, create 6 octets of keystream
45
46
    and output only what is needed
47
48
49
       else
50
          for (i = 0; i < 6; i++)
51
              maskSix[i] = csync[i % 4];
52
53
           /* Prevents cross-replay effects with other content types */
54
          for (i = 0; i < 4; i++)
55
              csync[i] ^= RlpContent;
56
57
58
       SCEMA (maskSix, SixOctets, csync, KSGArchitecture, KSGIdMask, ENCRYPTING,
                 dtcScheds + keyGenerator - 1);
```

```
1
2
          for (i = 0; i < len; i++)
3
              mask[i] = maskSix[i];
       }
8
10
```

3. Test Vectors

3.1. CAVE Test Vectors

These two test cases utilize the following fixed input data (expressed in hexadecimal form):

RANDSSD	=	4D	18EE	AA05	895C
Authentication Algorithm Version	=				C7
AUTH_DATA	=			79	2971
ESN	=			D75A	96EC
msg_buf[0] msg_buf[5]	=	B6, 2D, A2, 44, FE, 9B			

5

6

7

8

10

12

13

14

15

16

17

20

21

The following A-key and check digits should be entered in decimal form:

14 1421 3562 3730 9504 8808 6500

Conversion of the A-key, check digit entry into hex form will produce:

A-key, check bits = C442 F56B E9E1 7158, 1 51E4

The above entry, when combined with RANDSSD, will generate:

SSD_A = CC38 1294 9F4D CD0D SSD_B = 3105 0234 580E 63B4

3.1.1. Vector 1

If RAND_CHALLENGE = 34A2 B05F:

 $(Using SSD_AUTH = SSD_A)$

AUTH_SIGNATURE= 3 66F6

CMEA key k0, k7 = A0 7B 1C D1 02 75 69 14

ECMEA key = 5D ED AD 53 5B 4A B9 FC

offset_key = BD 71 D5 CD

SEED_NF key = 2F 15 F6 D1 27

ECMEA_NF key = 73 03 44 3C 55 DF B2 58

offset_nf_key = 14 6F 91 5B

sync = 3D A2

CMEA output = E5 6B 5F 01 65 C6

26 27

```
Mobile station:
2
                               ECMEA Output = d5 39 d7 45 cd 11
3
                               ECMEA NF Output = 3a 30 6a 40 39 b5
                               Base Station:
6
                               ECMEA Output = 50 9d c7 9b 19 d1
8
                               ECMEA NF Output = 96 7c 7b e4 9d 34
9
10
                               VPM = 18 93 94 82 4A 1A 2F 99
11
12
                                      A5 39 F9 5B 4D 22 D5 7C
                                      EE 32 AC 21 6B 26 0D 36
13
                                      A7 C9 63 88 57 8C B9 57
14
                                      E2 D6 CA 1D 77 B6 1F D5
15
16
                                      C7 1A 73 A4 17 B2 12 1E
                                      95 34 70 E3 9B CA 3F D0
17
18
                                      50 BE 4F D6 47 80 CC B8
                                      DF
19
20
                  3.1.2. Vector 2
21
                               If RAND CHALLENGE = 5375 DF99:
22
                               (Using SSD AUTH = SSD A)
23
                               AUTH SIGNATURE  0 255A
24
                               CMEA\ key\ k0, .k7 = F0 06 A8 5A 05 CD B3 2A
25
                               ECMEA key = B6 DF 9A D0 6E 5A 3D 14
26
                               offset_key = F9 A4 2C FA

SEED_NF key = 65 33 AE 92 C7
27
28
                               ECMEA NF key = 5C EF 0E E0 80 6A 1F 6B
29
                               offset nf key = C4 74 3C 71
30
31
                               sync = FF FF
32
33
                               CMEA output
                                                = 2B AD 16 A9 8F 32
34
                               Mobile station:
35
36
                               ECMEA Output = 91 cf e3 25 5e 44
37
                               ECMEA NF Output = cd 51 22 b2 74 49
38
39
                               Base Station:
40
41
                               ECMEA Output = f0 b8 9a 4b 06 55
42
                               ECMEA NF Output = 0d fb 93 e4 59 da
43
44
```

```
VPM = 20 38 01 6B 89 3C F8 A0
                                      28 48 98 75 AB 18 65 5A
2
                                      49 6E 0B BB D2 CB A8 28
3
                                      46 E6 D5 B4 12 B3 8C 9E
                                      76 6C 9E D4 98 C8 A1 4A
                                      D2 DC 94 B0 F6 D4 3E E0
6
                                      D1 6C 7E 9E AC 6B CA 43
                                      02 C9 23 63 6F 61 68 E8
8
9
                                      8F
                  3.1.3. Vector 3
10
                               If RAND CHALLENGE = 6c00 	ext{ } 0258:
11
                               (Using SSD AUTH = SSD A)
12
                               AUTH SIGNATURE = 0 8a8a
13
                               CMEA\ key\ k0,. .k7 = 5A C8 04 25 32 FB 2D 54
14
                               ECMEA key
                                                 = 20 64 57 F6 EE 60 EB AD
15
                               offset key
                                                 = E9 83 41 FB
16
                               SEED NF key
                                                 = 84 AD CF 40 BB
17
                               ECMEA_NF key = 33 37 C8 F3 85 50 C7 03 offset_nf_key = E0 2C 66 FA
18
19
20
                               sync = FF FF
21
                               CMEA output
22
                                                   = A3 06 25 D8 3E 21
23
                               Mobile station:
24
25
26
                               ECMEA Output = 41 ed 74 99 7d 41
                               ECMEA NF Output = ab aa 88 7e b6 f3
27
28
                               Base Station:
29
31
                               ECMEA Output = 6d 73 27 54 3d 9c
                               ECMEA NF Output = 8c e1 e2 b4 fd 62
32
33
34
                               VPM = ED A7 AA 63 27 EA F8 3D
                                      30 26 8C C5 18 88 8F 6D
35
                                      CD 0D 1D 97 21 06 2D 91
36
                                      1D CF 47 1F DD BE E3 E1
37
                                      71 18 26 73 7A 5F 09 CC
38
39
                                      13 2A 51 69 27 55 2B 2B
                                      OB 30 5A 09 F6 15 8F A7
40
                                      A9 55 7A 00 23 D8 FD 4C
41
                                      3E
42
```

3.1.4. Test Program

```
#include <stdio.h>
2
     #include "cave.h" /* see Exhibit 2-2 */
#include "ecmea.h" /* see Exhibit 2-29 */
3
     /* NAM stored data */
6
     unsigned char ESN[4] = \{ 0xd7, 0x5a, 0x96, 0xec \};
8
     unsigned char MIN1[3] = \{ 0x79, 0x29, 0x71 \};
9
     unsigned char A key[8];
10
     unsigned char SSD A NEW[8], SSD A[8];
11
     unsigned char SSD_B_NEW[8], SSD_B[8];
12
13
     /* data received from the network */
14
15
     unsigned char RANDSSD[7] = \{0x4d, 0x18, 0xee, 0xaa,
16
                                          0x05, 0x89, 0x5c};
17
     unsigned char RAND1[4] = \{0x34, 0xa2, 0xb0, 0x5f\};
18
    unsigned char RAND2[4] = \{0x53, 0x75, 0xdf, 0x99\};
unsigned char RAND3[4] = \{0x6c, 0x00, 0x02, 0x58\};
19
20
21
     /* cryptosync (meaning is air interface specific) */
22
23
     unsigned char sync1[2] = \{ 0x3d, 0xa2 \};
24
     unsigned char sync2[2] = { 0xff, 0xff };
25
26
     /* test plaintext */
27
28
     unsigned char buf[6] = { 0xb6, 0x2d, 0xa2, 0x44, 0xfe, 0x9b };
29
30
     /* entered A key and checksum */
31
32
     char digits[26] =
33
         { '1', '4', '1', '4', '2', '1', '3', '5', '6', '2', '3', '7', '3', '0', '9', '5', '0', '4', '8', '8', '0', '8', '6', '5', '0', '0' };
34
35
36
37
     void pause(void)
38
39
        printf("Enter to continue\n");
40
        getchar();
41
42
43
```

```
void main(void)
1
2
        int i, j;
3
        unsigned char auth_data[3], test buf[6];
        unsigned long AUTHR;
5
6
        /* check A key and SSD */
7
8
        if(A_Key_Verify(digits))
9
10
           printf("A key verified ok\n");
11
12
13
        else
14
           printf("A key verification failed\n");
15
           return;
16
17
18
        /* check SSD generation process */
19
20
        SSD Generation(RANDSSD);
21
22
        SSD Update();
23
        printf("SSD A =");
24
        for (i = 0; i < 4; i++)
25
26
           printf(" ");
27
           for (j = 0; j < 2; j++)
28
              printf("%02x",(unsigned int)SSD_A[2*i+j]);
29
30
        printf("\n");
31
32
        printf("SSD B =");
33
        for (i = 0; i < 4; i++)
34
35
           printf(" ");
36
           for (j = 0; j < 2; j++)
37
              printf("%02x",(unsigned int)SSD_B[2*i+j]);
38
39
       printf("\n");
40
41
        /* Inputs for test vectors */
42
43
        /* put MIN1 into auth data (no dialed digits for this test) */
44
45
        for (i = 0; i < 3; i++)
46
           auth data[i] = MIN1[i];
47
48
        /* vector 1 */
49
50
        printf("\nVector 1\n\n");
51
52
        AUTHR = Auth_Signature(RAND1, auth_data, SSD_A, 1);
53
54
        printf("RAND CHALLENGE =");
55
        for (i = 0; \overline{i} < 2; i++)
56
57
           printf(" ");
58
           for (j = 0; j < 2; j++)
59
              printf("%02x",(unsigned int)RAND1[2*i+j]);
60
61
```

```
printf("\n");
1
       printf("AUTH SIGNATURE = %01lx %04lx\n", AUTHR >> 16,
3
          AUTHR & 0 \times 00000 \text{ ffff});
       for (i = 0; i < 6; i++)
6
           test buf[i] = buf[i];
8
       Key VPM Generation();
9
       ECMEA Secret Generation();
10
       Non Financial Seed Key Generation();
11
       Non Financial Secret Generation();
12
13
       printf("
                     CMEA key =");
14
       for (i = 0; i < 8; i++)
15
           printf(" %02x", (unsigned int)cmeakey[i]);
16
       printf("\n");
17
18
                    ECMEA key =");
       printf("
19
       for (i = 0; i < 8; i++)
20
          printf(" %02x",(unsigned int)ecmea_key[i]);
21
22
       printf("\n");
23
       printf("
                 offset key =");
24
       for (i = 0; i < 4; i++)
25
           printf(" %02x",(unsigned int)offset key[i]);
26
27
       printf("\n");
28
       printf("
                 SEED NF key =");
29
       for (i = 0; i < 5; i++)
30
          printf(" %02x", (unsigned int) seed nf key[i]);
31
       printf("\n");
32
33
       printf(" ECMEA NF key =");
34
       for (i = 0; i < 8; i++)
35
           printf(" %02x", (unsigned int)ecmea_nf_key[i]);
36
       printf("\n");
37
38
       printf(" offset_nf_key =");
39
       for (i = 0; i < 4; i++)
40
           printf(" %02x",(unsigned int)offset_nf_key[i]);
41
       printf("\n");
42
43
       printf("
44
                   sync =");
       printf(" %02x %02x\n", (unsigned int)sync1[0],
45
           (unsigned int)sync1[1]);
46
47
       pause();
48
49
       printf("
                           Input =");
50
       for (i = 0; i < 6; i++)
51
          printf(" %02x", (unsigned int)test buf[i]);
52
       printf("\n");
53
54
55
       CMEA(test_buf,6);
56
                   CMEA Output =");
       printf("
57
       for (i = 0; i < 6; i++)
58
           printf(" %02x",(unsigned int)test_buf[i]);
59
       printf("\n");
60
61
```

```
for (i = 0; i < 6; i++)
1
           test buf[i] = buf[i];
2
        ECMEA(test buf, 6, sync1, 0, 0);
3
       printf("
                  ECMEA Output =");
       for (i = 0; i < 6; i++)
6
           printf(" %02x", (unsigned int)test buf[i]);
       printf("\n");
8
9
       for (i = 0; i < 6; i++)
10
           test buf[i] = buf[i];
11
       ECMEA(test buf, 6, sync1, 0, 1);
12
13
       printf("ECMEA NF Output =");
14
        for (i = 0; i < 6; i++)
15
           printf(" %02x",(unsigned int)test buf[i]);
16
       printf("\n");
17
18
       printf("VPM =");
19
       for (i = 0; i < 65; i++)
20
21
           printf(" %02x", (unsigned int) VPM[i]);
22
           if (((i+1)%8) == 0)
23
                             ");
              printf("\n
24
25
       printf("\n");
26
27
       pause();
28
29
        /* vector 2 */
30
31
       printf("\nVector 2\n\n");
32
33
       AUTHR = Auth Signature(RAND2, auth data, SSD A, 1);
34
35
        printf("RAND CHALLENGE =");
36
        for (i = 0; \overline{i} < 2; i++)
37
38
           printf(" ");
39
           for (j = 0; j < 2; j++)
40
              printf("%02x",(unsigned int)RAND2[2*i+j]);
41
42
43
       printf("\n");
44
       printf("AUTH SIGNATURE = %01lx %04lx\n", AUTHR >> 16,
45
           AUTHR & 0x0000ffff);
46
47
        for (i = 0; i < 6; i++)
48
           test buf[i] = buf[i];
49
50
        Key VPM Generation();
51
        ECMEA Secret Generation();
52
       Non Financial Seed Key Generation();
53
       Non_Financial_Secret_Generation();
54
55
       printf("
                      CMEA key =");
56
       for (i = 0; i < 8; i++)
57
           printf(" %02x",(unsigned int)cmeakey[i]);
58
       printf("\n");
59
60
       printf("
                      ECMEA key =");
61
```

```
for (i = 0; i < 8; i++)
1
           printf(" %02x",(unsigned int)ecmea key[i]);
2
       printf("\n");
3
       printf("
                  offset key =");
5
       for (i = 0; i < 4; i++)
6
          printf(" %02x", (unsigned int)offset key[i]);
7
       printf("\n");
8
9
       printf(" SEED NF key =");
10
       for (i = 0; i < 5; i++)
11
           printf(" %02x", (unsigned int) seed nf key[i]);
12
       printf("\n");
13
14
       printf(" ECMEA NF key =");
15
       for (i = 0; i < 8; i++)
16
          printf(" %02x", (unsigned int)ecmea nf key[i]);
17
       printf("\n");
18
19
       printf(" offset nf key =");
20
       for (i = 0; i < 4; i++)
21
           printf(" %02x", (unsigned int)offset nf key[i]);
22
       printf("\n");
23
24
       printf("
                    sync =");
25
       printf(" %02x %02x\n", (unsigned int)sync2[0],
26
27
           (unsigned int)sync2[1]);
28
       pause();
29
30
31
       printf("
                           Input =");
       for (i = 0; i < 6; i++)
32
          printf(" %02x",(unsigned int)test_buf[i]);
33
       printf("\n");
34
35
       CMEA(test buf, 6);
36
37
38
       printf("
                  CMEA Output =");
       for (i = 0; i < 6; i++)
39
           printf(" %02x",(unsigned int)test_buf[i]);
40
       printf("\n");
41
42
43
       for (i = 0; i < 6; i++)
           test buf[i] = buf[i];
44
       ECMEA(test buf, 6, sync2, 0, 0);
45
46
       printf("
                  ECMEA Output =");
47
       for (i = 0; i < 6; i++)
48
          printf(" %02x",(unsigned int)test_buf[i]);
49
       printf("\n");
50
51
       for (i = 0; i < 6; i++)
52
           test buf[i] = buf[i];
53
       ECMEA(test buf, 6, sync2, 0, 1);
54
55
       printf("ECMEA NF Output =");
56
       for (i = 0; i < 6; i++)
57
           printf(" %02x", (unsigned int)test buf[i]);
58
       printf("\n");
59
60
       printf("VPM =");
61
```

```
for (i = 0; i < 65; i++)
1
2
           printf(" %02x",(unsigned int)VPM[i]);
3
           if (((i+1)%8) == 0)
              printf("\n
5
6
       printf("\n");
7
8
       pause();
9
10
       /* vector 3 */
11
12
       printf("\nVector 3\n\n");
13
14
       AUTHR = Auth Signature(RAND3, auth data, SSD A, 1);
15
16
       printf("RAND CHALLENGE =");
17
       for (i = 0; \overline{i} < 2; i++)
18
19
           printf(" ");
20
           for (j = 0; j < 2; j++)
21
              printf("%02x",(unsigned int)RAND3[2*i+j]);
22
23
       printf("\n");
24
25
       printf("AUTH SIGNATURE = %01lx %04lx\n", AUTHR >> 16,
26
27
          AUTHR & 0x0000ffff);
28
       for (i = 0; i < 6; i++)
29
           test buf[i] = buf[i];
30
31
       Key VPM Generation();
32
       ECMEA Secret Generation();
33
       Non Financial Seed Key Generation();
34
       Non Financial Secret Generation();
35
36
       printf("
                      CMEA key =");
37
       for (i = 0; i < 8; i++)
38
           printf(" %02x",(unsigned int)cmeakey[i]);
39
       printf("\n");
40
41
       printf(" ECMEA key =");
42
       for (i = 0; i < 8; i++)
43
           printf(" %02x", (unsigned int)ecmea key[i]);
44
       printf("\n");
45
46
       printf("
                   offset key =");
47
       for (i = 0; i < 4; i++)
48
          printf(" %02x", (unsigned int)offset_key[i]);
49
       printf("\n");
50
51
       printf(" SEED NF key =");
52
       for (i = 0; i < 5; i++)
53
           printf(" %02x", (unsigned int)seed_nf_key[i]);
54
55
       printf("\n");
56
       printf(" ECMEA NF key =");
57
       for (i = 0; i < 8; i++)
58
           printf(" %02x", (unsigned int)ecmea nf key[i]);
59
60
       printf("\n");
61
```

```
printf(" offset nf key =");
1
       for (i = 0; i < 4; i++)
2
           printf(" %02x", (unsigned int)offset_nf_key[i]);
3
       printf("\n");
                   sync =");
6
       printf("
       printf(" %02x %02x\n", (unsigned int)sync2[0],
7
           (unsigned int)sync2[1]);
8
9
       pause();
10
11
       printf("
                           Input =");
12
       for (i = 0; i < 6; i++)
13
          printf(" %02x",(unsigned int)test_buf[i]);
14
       printf("\n");
15
16
       CMEA(test_buf,6);
17
18
       printf("
                 CMEA Output =");
19
       for (i = 0; i < 6; i++)
20
          printf(" %02x",(unsigned int)test_buf[i]);
21
22
       printf("\n");
23
       for (i = 0; i < 6; i++)
24
           test buf[i] = buf[i];
25
26
       ECMEA(test buf,6,sync2,0,0);
27
       printf(" ECMEA Output =");
28
       for (i = 0; i < 6; i++)
29
          printf(" %02x", (unsigned int)test buf[i]);
30
31
       printf("\n");
32
33
       for (i = 0; i < 6; i++)
          test buf[i] = buf[i];
34
       ECMEA(test buf, 6, sync2, 0, 1);
35
36
       printf("ECMEA NF Output =");
37
       for (i = 0; i < 6; i++)
38
          printf(" %02x",(unsigned int)test_buf[i]);
39
       printf("\n");
40
41
       printf("VPM =");
42
43
       for (i = 0; i < 65; i++)
44
           printf(" %02x", (unsigned int) VPM[i]);
45
           if (((i+1)%8) == 0)
46
              printf("\n ");
47
48
       printf("\n");
49
50
51
       pause();
52
53
```

3.2. Wireless Residential Extension Test Vector

2	3.2.1. Input data
3 4 5	Manufacturer's Key = 2 14 0E 9F 70 50 D7 EA 42 D9 C9 00 C9 14 14 CF
6	BID = 00 00 01 00
7	Random Challenge = 7E 49 AE 4F
8	ACRE Phone Number = 549-8506
9	Random WRE = 3 17 52
10	ESN = ED 07 13 95
11	Random WIKEY = B7 FC 75 5A F0 A4 90
12	WRE Key = CB 60 F9 9F 5B 15 6F AE

3.2.2. Test Program

```
#include <stdio.h>
2
    #include "cave.h" /* see Exhibit 2-2 */
3
    #include "wre.h"
                        /* see Exhibit 2-31 */
    /* NAM stored data */
6
    unsigned char ESN[4] = \{ 0xd7, 0x5a, 0x96, 0xec \};
8
    unsigned char MIN1[3] = \{ 0x79, 0x29, 0x71 \};
9
    unsigned char A key[8];
10
    unsigned char SSD A NEW[8], SSD A[8];
11
    unsigned char SSD_B_NEW[8], SSD_B[8];
12
13
14
    /* Test vector inputs */
15
16
    unsigned char manufact[16] = \{0x85, 0x03, 0xA7, 0xDC,
17
                                      0x14, 0x35, 0xFA, 0x90,
18
                                      0xB6, 0x72, 0x40, 0x32,
19
                                      0x45, 0x05, 0x33, 0xC0 };
20
21
    unsigned char baseid[4] = { 0x00, 0x00, 0x01, 0x00 };
22
23
    unsigned char random challenge[4] = \{0x7E, 0x49, 0xAE, 0x4F\};
24
25
    unsigned char acre phone [3] = \{ 0x49, 0x85, 0xA6 \};
26
27
    unsigned char random wre[3] = \{0x62, 0xEA, 0x40\};
28
29
    unsigned char hs esn[4] = \{ 0xED, 0x07, 0x13, 0x95 \};
30
31
    unsigned char rand wikey[7] = \{ 0xB7, 0xFC, 0x75, 0x5A, 
32
                                       0xF0, 0xA4, 0x90 };
33
34
    /* CAVE outputs */
35
36
    extern unsigned char
37
                              WIKEY[8];
    extern unsigned char
                              WIKEY NEW[8];
38
                              WRE KEY[8];
    extern unsigned char
39
40
    void main(void)
41
42
        int i;
43
        unsigned long auth sig;
44
45
        WIKEY Generation (manufact, baseid);
46
        print\overline{f}("WIKEY = ");
47
        for(i=0;i<8;i++)
48
           printf("%02x",(unsigned int)WIKEY[i]);
49
        printf("\n");
50
51
        auth sig = WI Auth Signature(random challenge,baseid,acre phone);
52
53
        printf("AUTH SIGNATURE = %05lx\n", auth sig);
54
        WRE_KEY[0] = 0xCB;
55
        WRE KEY[1] = 0x60;
56
        WRE KEY[2] = 0xF9;
57
        WRE KEY[3] = 0x9F;
58
        WRE KEY[4] = 0x5B;
```

18

19

20

21

22

```
WRE KEY[5] = 0x15;
1
        WRE KEY[6] = 0x6F;
2
        WRE KEY[7] = 0xAE;
3
        auth_sig = WRE_Auth_Signature(random_wre,baseid,hs_esn);
        printf("AUTH SIGNATURE = %05lx\n", auth_sig);
6
        WIKEY Update(rand wikey,baseid);
8
        printf("WIKEY NEW = ");
9
10
        for(i=0;i<8;i++)
          printf("%02x",(unsigned int)WIKEY NEW[i]);
11
        printf("\n");
12
13
       printf("Enter to exit\n");
14
       getchar();
15
16
17
```

3.2.3. Test Program Output

```
WIKEY = cb60f99f5b156fae

AUTH_SIGNATURE = 2cf01

AUTH_SIGNATURE = 12893

WIKEY_NEW = 167ca928358cceba
```

3.3. Basic Data Encryption Test Vector

3.3.1. Input data

2

```
SSD_B= 1492 5280 1776 1867

RAND = 1234 ABCD

HOOK = CDEF 5678

24 octets of mask to be returned
```

3.3.2. Test Program

```
8
    #include <stdio.h>
    #include "cave.h" /* see Exhibit 2-2 */
9
    #include "oryx.h"
                        /* see Exhibit 2-45 */
10
11
    /* NAM stored data */
12
13
    unsigned char ESN[4] = \{ 0xd7, 0x5a, 0x96, 0xec \};
14
    unsigned char MIN1[3] = \{ 0x79, 0x29, 0x71 \};
15
16
    unsigned char A key[8];
17
    unsigned char SSD A NEW[8], SSD A[8];
    unsigned char SSD B NEW[8], SSD B[8];
18
19
20
    void pause(void)
21
22
       printf("Enter to continue\n");
23
       getchar();
24
25
26
    void main(void)
27
       int i, j;
28
       unsigned long hook;
29
       unsigned char buf[24], rand[4];
30
31
       rand[0] = 0x12;
32
33
       rand[1] = 0x34;
       rand[2] = 0xab;
       rand[3] = 0xcd;
35
       hook = 0xcdef5678;
37
38
39
       SSD B[0] = 0x14;
       SSD B[1] = 0x92;
40
41
       SSD B[2] = 0x52;
42
       SSD B[3] = 0x80;
43
       SSD B[4] = 0x17;
44
       SSD B[5] = 0x76;
       SSD B[6] = 0x18;
45
       SSD B[7] = 0x67;
46
47
       printf("\nSSD B =");
48
49
       for (i = 0; i < 4; i++)
50
           printf(" ");
51
           for (j = 0; j < 2; j++)
52
53
              printf("%02x", (unsigned int)SSD B[2*i+j]);
54
```

```
1
        }
2
3
        printf("\nRAND =");
5
        for (i = 0; i < 2; i++)
6
           printf(" ");
7
           for (j = 0; j < 2; j++)
8
9
              printf("%02x", (unsigned int)rand[2*i+j]);
10
11
        }
12
13
        printf("\nHOOK = %04lx %04lx\n", hook >> 16, hook & 0x0000ffff);
14
15
       pause();
16
17
        printf("24 octets of mask to be returned");
18
19
        DataKey = DataKey Generation();
20
21
        printf("\n\nOutput:\n\n");
22
23
        printf("\nDataKey = %04lx %04lx\n", DataKey >> 16,
24
           DataKey & 0x0000ffff);
25
26
        LTable_Generation(rand);
27
28
        printf("\n\nL:\n\n");
29
30
        for(i = 0; i < 16; i++)
31
32
           for (j = 0; j < 16; j++)
33
34
              printf("%02x ", (unsigned int)L[16*i+j]);
35
36
           printf("\n");
37
        }
38
39
       pause();
40
41
        Data Mask (DataKey, hook, 24, buf);
42
43
        printf("\n\nmask:\n\n");
44
45
        for(i = 0; i < 2; i++)
46
47
           for (j = 0; j < 12; j++)
48
49
              printf("%02x ", (unsigned int)buf[12*i+j]);
50
51
           printf("\n");
52
53
54
        pause();
55
56
```

3.3.3. Test Program Output

DataKey = 8469 B522 47 D1 88 BC 3B 7F 25 30 16 CE A9 9D FF FB 2F E4 15 83 04 A3 96 1F 09 B6 A7 70 29 D2 2E 60 2B 5A 6C 66 33 53 7B DE 2D 20 F1 8C 4F E5 93 39 8E 6A 13 06 62 FD 0C 6F 0E 0F 4D 3D 14 32 A1 50 E2 1B 69 6B 79 40 36 5D E8 74 FC B8 51 10 D9 F2 CB 5E C5 86 6D F0 2C 65 7D 5F 8B BE 8F DA B4 4A BA 64 4E 76 00 9F 7E 07 49 48 95 75 71 6E CC 68 38 0D 10 17 A8 78 46 90 C0 41 BF 94 97 D3 43 01 C8 AB DD 11 8A 1C BB 08 F6 4C 4B 27 28 1A 03 C4 FA E7 B5 A2 12 EB B3 C9 72 52 A0 0A E9 D8 C6 3F AF 05 CA C3 AE 13 9E 9A EF B7 8D E6 A4 D5 82 F3 77 54 42 B2 18 73 14 E1 DC BD B9 3E 37 59 CD EC 02 80 81 AC 2A 31 EA 15 89 1E 63 D6 91 92 D4 11 EE 9C 12 A5 A6 3A C2 35 16 F5 67 CF 45 44 DB 22 FE 55 C7 56 B1 AD F4 F9 57 17 F8 DF 1D 58 9B 34 ED 0B D7 AA 99 7A C1 7C E0 E3 18 5B 5C 21 61 85 19 84 D0 3C 26 87 98 B0 F7 23 24 19 20 21 mask 57 F6 C2 03 7C 78 2F CC 8B 3E E4 0B 22 E0 4D 73 80 FF 2A 4D 2F 8D 74 8E DB 23

17

18

3.4. Enhanced Voice and Data Privacy Test Vectors

3.4.1. Input Data

```
Data buffer = B6 2D A2 44 FE 9B 23 AB

Vector 1:

CMEA key k0,. .k7 = a0 7b 1c d1 02 75 69 14

sync = 3d 00 a2 00

Vector 2:

CMEA key k0,. .k7 = F0 06 A8 5A 05 CD B3 2A

sync = ff 00 ff 00
```

3.4.2. Test Program

3.4.2.1. Main program file

```
19
                 EPE test file "main.c"
20
21
                 Explicitly contains code for generating vector sets 1 (DTC key
22
                 schedule) and 2 (DCCH key schedule). These first two sets also test
23
                 SCEMA. The key schedules are needed for generating the remaining
24
                 vector sets. However, none of the remaining sets depend upon other sets
25
                being generated.
26
                 */
27
28
                 #include <stdio.h>
29
30
                 #include "cave.h"
31
32
                 #include "scema.h"
33
34
                 void pause(void)
35
36
                             printf("Enter to continue\n");
37
                            getchar();
38
39
40
41
                 void main(void)
42
                             unsigned int i;
43
44
                 /* test plaintext */
45
46
47
                             const unsigned char buf[8] = \{0xb6, 0x2d, 0xa2, 0x44, 0x44
                                                                                                                                                      0xfe,0x9b,0x23, 0xab);
48
                             unsigned char testBuf[MaxMessageOctetSize];
49
                             unsigned char testBufTwo[MaxFrameOctetSize];
50
```

```
1
       /* cryptosync (meaning is air interface specific) */
       unsigned char sync1[4] = \{0x3d,0x00,0xa2,0x00\};
       unsigned char sync2[4] = \{0xff,0x00,0xff,0x00\};
5
6
       /* vector set 1 */
8
       cmeakey[0] = 0xA0;
9
       cmeakey[1] = 0x7B;
10
       cmeakey[2] = 0x1C;
11
       cmeakey[3] = 0xD1;
12
       cmeakey[4] = 0x02;
13
       cmeakey[5] = 0x75;
14
       cmeakey[6] = 0x69;
15
       cmeakey[7] = 0x14;
16
17
       printf("\nVector Set 1 - DTC Key Generation and SCEMA\n\n");
18
19
       DTC Key Generation();
20
21
22
       printf("
                                  DTC CMEA key =");
       for (i = 0; i < 8; i++)
23
           printf(" %02x",(unsigned int)cmeakey[i]);
24
       printf("\n");
25
26
27
       printf("
                    DTC scemaKey (CaveKey1) =");
       for (i = 0; i < 8; i++)
28
           printf(" %02x", (unsigned int) (dtcScheds) ->scemaKey[i]);
29
       printf("\n");
30
31
       printf("
                                          sync =");
32
       printf(" %02x %02x %02x %02x\n", (unsigned int)sync1[0],
33
           (unsigned int)sync1[1], (unsigned int)sync1[2],
34
           (unsigned int)sync1[3]);
35
36
       for (i = 0; i < SixOctets; i++)
37
           testBuf[i] = buf[i];
38
39
       printf("
                                         Input =");
40
       for (i = 0; i < SixOctets; i++)
41
           printf(" %02x", (unsigned int)testBuf[i]);
42
43
       printf("\n");
44
       SCEMA(testBuf,SixOctets,sync1,0,0,ENCRYPTING,dtcScheds);
45
46
                             DTC SCEMA Output =");
47
       for (i = 0; i < SixOctets; i++)
48
           printf(" %02x", (unsigned int)testBuf[i]);
49
       printf("\n");
50
51
       pause();
52
53
       /* vector set 2 */
54
55
       cmeakey[0] = 0xf0;
56
       cmeakey[1] = 0x06;
57
       cmeakey[2] = 0xa8;
58
       cmeakey[3] = 0x5a;
59
       cmeakey[4] = 0x05;
60
       cmeakey[5] = 0xcd;
61
```

```
cmeakey[6] = 0xb3;
1
       cmeakey[7] = 0x2a;
2
       printf("\nVector Set 2 - DCCH Key Generation and SCEMA\n\n");
       DCCH Key Generation();
6
       printf("
                                 DCCH CMEA key =");
8
       for (i = 0; i < 8; i++)
9
          printf(" %02x", (unsigned int)cmeakey[i]);
10
       printf("\n");
11
12
       printf("
                     DCCH scemaKey (CaveKey1) = ");
13
       for (i = 0; i < 8; i++)
14
          printf(" %02x", (unsigned int) (dcchScheds) ->scemaKey[i]);
15
       printf("\n");
16
17
       printf("
                                           sync =");
18
       printf(" %02x %02x %02x %02x\n", (unsigned int)sync2[0],
19
           (unsigned int)sync2[1], (unsigned int)sync2[2],
20
           (unsigned int)sync2[3]);
21
22
       for (i = 0; i < SixOctets; i++)
23
          testBuf[i] = buf[i];
24
25
       printf("
                                          Input =");
26
       for (i = 0; i < SixOctets; i++)
27
          printf(" %02x", (unsigned int)testBuf[i]);
28
       printf("\n");
29
30
       SCEMA (testBuf, SixOctets, sync2, 0, 0, ENCRYPTING, dcchScheds);
31
32
                             DCCH SCEMA Output =");
33
       for (i = 0; i < SixOctets; i++)
34
          printf(" %02x", (unsigned int)testBuf[i]);
35
       printf("\n");
36
37
38
       pause();
39
    Note: None of these remaining tests are mutually dependent, and can
40
    thus be selectively disabled.
41
42
43
    /* Vector Set 3 - SCEMA KSG */
    #include "vs3scemaKSG.h"
44
45
    /* Vector Set 4 - Long Block Encryptor */
46
    #include "vs4longBlock.h"
47
48
    /* Vector Set 5 - Short Block Encryptor */
49
    #include "vs5shortBlock.h"
50
51
    /* Vector Set 6 - Enhanced Message Encryption */
52
    #include "vs6enhMsqEnc.h"
53
54
    /* Vector Set 7 - Enhanced Voice Privacy */
55
    #include "vs7enhVoicePriv.h"
56
57
    /* Vector Set 8 - Enhanced Data Mask Generation */
58
    #include "vs8enhDataMask.h"
59
60
61
```

3.4.2.2. Vector set 3

1

```
2
    /* Vector Set 3 - SCEMA KSG "vs3scemaKSG.h" */
3
       printf("\nVector Set 3 - SCEMA KSG\n\n");
5
       /* Voice content, Reverse Channel, 3-octet input, 8-octet output */
       printf("\nVoice content, Reverse Channel, 3-octet input, 8-octet
8
    output\n\n");
9
10
       for (i = 0; i < ThreeOctets; i++)
11
           testBuf[i] = buf[i];
12
13
       printf("
                            Input =");
14
       for (i = 0; i < ThreeOctets; i++)</pre>
15
           printf(" %02x", (unsigned int)testBuf[i]);
16
       printf("\n");
17
18
19
       SCEMA KSG(testBufTwo, EightOctets, testBuf, ThreeOctets,
20
                 VoiceContent,dtcScheds,ReverseChannel);
21
22
23
       printf("SCEMA KSG Output =");
24
       for (i = 0; i < EightOctets; i++)</pre>
25
           printf(" %02x", (unsigned int)testBufTwo[i]);
26
       printf("\n\n");
27
28
29
        /* Voice content, Reverse Channel, 6-octet input, 6-octet output */
30
       printf("\nVoice content, Reverse Channel, 6-octet input, 6-octet
31
    output\n\n");
32
33
       for (i = 0; i < SixOctets; i++)
34
           testBuf[i] = buf[i];
35
36
       printf("
                            Input =");
37
       for (i = 0; i < SixOctets; i++)
38
           printf(" %02x", (unsigned int)testBuf[i]);
39
       printf("\n");
40
41
42
       SCEMA KSG(testBufTwo, SixOctets, testBuf, SixOctets,
43
                 VoiceContent,dtcScheds,ReverseChannel);
44
45
46
       printf("SCEMA KSG Output =");
47
       for (i = 0; i < SixOctets; i++)
48
           printf(" %02x", (unsigned int)testBufTwo[i]);
49
       printf("\n\n");
50
51
52
53
       Voice content, Reverse Channel, 6-octet input,
54
       3-octet requested output, 6 octets delivered
55
56
       printf("\nVoice content, Reverse Channel, 6-octet input,\n");
57
       printf(" 3-octet requested output, 6-octets delivered\n\n");
58
59
       for (i = 0; i < SixOctets; i++)
60
```

```
testBuf[i] = buf[i];
1
2
       printf("
                            Input =");
3
       for (i = 0; i < SixOctets; i++)
          printf(" %02x", (unsigned int)testBuf[i]);
6
       printf("\n");
8
       SCEMA KSG(testBufTwo, ThreeOctets, testBuf, SixOctets,
9
                 VoiceContent,dtcScheds,ReverseChannel);
10
11
12
       printf("SCEMA KSG Output =");
13
       for (i = 0; i < SixOctets; i++)
14
          printf(" %02x", (unsigned int)testBufTwo[i]);
15
       printf("\n\n");
16
17
       pause();
18
19
20
       printf("\nVector Set 3 - SCEMA KSG cont'd\n\n");
21
22
       /* Message content, Reverse Channel, 6-octet input, 6-octet output */
23
       printf("\nMessage content, Reverse Channel, 6-octet input, 6-octet
24
    output\n');
25
26
       for (i = 0; i < SixOctets; i++)
27
           testBuf[i] = buf[i];
28
29
       printf("
                            Input =");
30
       for (i = 0; i < SixOctets; i++)
31
          printf(" %02x", (unsigned int)testBuf[i]);
32
       printf("\n");
33
34
       SCEMA KSG(testBufTwo, SixOctets, testBuf, SixOctets,
35
                 MessageContent,dtcScheds,ReverseChannel);
36
37
       printf("SCEMA KSG Output =");
38
       for (i = 0; i < SixOctets; i++)
39
          printf(" %02x", (unsigned int)testBufTwo[i]);
40
       printf("\n\n");
41
42
43
       /* Message content, Forward Channel, 6-octet input, 6-octet output */
44
       printf("\nMessage content, Forward Channel, 6-octet input, 6-octet
45
    output\n\n";
46
47
       for (i = 0; i < SixOctets; i++)
48
          testBuf[i] = buf[i];
49
50
                            Input =");
51
       for (i = 0; i < SixOctets; i++)
52
          printf(" %02x",(unsigned int)testBuf[i]);
53
       printf("\n");
54
55
       SCEMA KSG(testBufTwo, SixOctets, testBuf, SixOctets,
56
                 MessageContent,dtcScheds,ForwardChannel);
57
58
       printf("SCEMA KSG Output =");
59
       for (i = 0; i < SixOctets; i++)
60
          printf(" %02x", (unsigned int)testBufTwo[i]);
61
```

```
printf("\n\n");
1
       pause();
                        3.4.2.3. Vector set 4
6
7
    /* Vector Set 4 - Long Block Encryptor "vs4longBlock.h" */
8
q
       printf("\nVector Set 4 - Long Block Encryptor\n\n");
10
11
12
        /* Encryption/Decryption (Voice content, Reverse Channel) */
       printf("\nEncryption/Decryption (Voice content, Reverse
13
    Channel)\n\n";
14
15
       for (i = 0; i < SixOctets; i++)
16
           testBuf[i] = buf[i];
17
18
       printf("
                                         Input =");
19
       for (i = 0; i < SixOctets; i++)
20
           printf(" %02x", (unsigned int)testBuf[i]);
21
       printf("\n");
22
23
24
       Long Block Encryptor(testBuf, VoiceContent, ENCRYPTING,
25
26
                           dtcScheds, ReverseChannel);
27
28
       printf("Long Block Encryptor Output =");
29
       for (i = 0; i < SixOctets; i++)
30
           printf(" %02x", (unsigned int)testBuf[i]);
31
       printf("\n");
32
33
34
       Long Block Encryptor(testBuf, VoiceContent, DECRYPTING,
35
                           dtcScheds,ReverseChannel);
36
37
38
       printf("Long Block Decryptor Output =");
       for (i = 0; i < SixOctets; i++)
39
           printf(" %02x", (unsigned int)testBuf[i]);
40
       printf("\n\n");
41
42
43
        /* Encryption (Message Content, Reverse Channel) */
44
45
       printf("\nEncryption (Message Content, Reverse Channel)\n\n");
46
       for (i = 0; i < SixOctets; i++)
47
           testBuf[i] = buf[i];
48
49
       printf("
                                         Input ="); for (i = 0; i < SixOctets;</pre>
50
51
    i++)
           printf(" %02x", (unsigned int)testBuf[i]);
52
       printf("\n");
53
54
55
       Long Block Encryptor(testBuf, MessageContent, ENCRYPTING,
56
                           dtcScheds,ReverseChannel);
57
58
```

```
printf("Long Block Encryptor Output =");
1
       for (i = 0; i < SixOctets; i++)
2
          printf(" %02x",(unsigned int)testBuf[i]);
3
       printf("\n\n");
6
       /* Encryption (Voice Content, Forward Channel) */
       printf("\nEncryption (Voice Content, Forward Channel)\n\n");
8
9
       for (i = 0; i < SixOctets; i++)
10
          testBuf[i] = buf[i];
11
12
       printf("
                                        Input =");
13
       for (i = 0; i < SixOctets; i++)
14
          printf(" %02x", (unsigned int)testBuf[i]);
15
       printf("\n");
16
17
18
       Long_Block_Encryptor(testBuf, VoiceContent, ENCRYPTING,
19
                           dtcScheds, ForwardChannel);
20
21
22
       printf("Long Block Encryptor Output =");
23
       for (i = 0; i < SixOctets; i++)
24
          printf(" %02x", (unsigned int)testBuf[i]);
25
       printf("\n\n");
26
27
       pause();
28
29
                        3.4.2.4. Vector set 5
30
31
    /* Vector Set 5 - Short Block Encryptor "vs5shortBlock.h"
32
33
    Note: The last octets of the decrypted buffers may not match the
34
    original input buffers' last octets. This is legitimate and comprises a
35
    test to ensure that the output clean up code is working to zero out non-
36
    content bearing bits.
37
38
    */
39
40
       printf("\n\nVector Set 5 - Short Block Encryptor\n");
41
42
       /* Encryption/Decryption (47 bits, Voice content, Reverse Channel) */
43
44
       printf("\nEncryption/Decryption (47 bits, Voice content, Reverse
45
    Channel) \n\n");
46
47
       for (i = 0; i < SixOctets; i++)
48
49
50
          testBuf[i] = buf[i];
           testBufTwo[i] = buf[i + 1];
51
52
53
       printf(" SB Data Mask Input =");
54
       for (i = 0; i < SixOctets; i++)
55
          printf(" %02x", (unsigned int)testBuf[i]);
56
       printf("\n");
57
58
```

```
Short Block Encryptor(testBuf, 47, VoiceContent, testBufTwo,
1
                           ENCRYPTING, dtcScheds, ReverseChannel);
2
3
       printf("SB Data Mask Output =");
       for (i = 0; i < SixOctets; i++)
5
           printf(" %02x", (unsigned int)testBuf[i]);
6
       printf("\n");
7
8
       Short Block Encryptor(testBuf, 47, VoiceContent, testBufTwo,
9
                           DECRYPTING, dtcScheds, ReverseChannel);
10
11
       printf("SB Data Mask Output =");
12
       for (i = 0; i < SixOctets; i++)
13
           printf(" %02x", (unsigned int)testBuf[i]);
14
       printf("\n");
15
16
17
       /* Encryption/Decryption (17 bits, Voice content, Reverse Channel) */
18
19
       printf("\nEncryption/Decryption (17 bits, Voice content,Reverse
20
    Channel\n\n");
21
22
       for (i = 0; i < SixOctets; i++)
23
24
           testBuf[i] = buf[i];
25
           testBufTwo[i] = buf[i + 1];
26
27
28
       printf(" SB Data Mask Input =");
29
       for (i = 0; i < SixOctets; i++)
30
           printf(" %02x", (unsigned int)testBuf[i]);
31
       printf("\n");
32
33
       Short_Block_Encryptor(testBuf,17,VoiceContent,testBufTwo,
34
                           ENCRYPTING, dtcScheds, ReverseChannel);
35
36
       printf("SB Data Mask Output =");
37
38
       for (i = 0; i < SixOctets; i++)
           printf(" %02x",(unsigned int)testBuf[i]);
39
       printf("\n");
40
41
       Short Block Encryptor(testBuf, 17, VoiceContent, testBufTwo,
42
43
                           DECRYPTING, dtcScheds, ReverseChannel);
44
       printf("SB Data Mask Output =");
45
       for (i = 0; i < SixOctets; i++)
46
           printf(" %02x", (unsigned int)testBuf[i]);
47
       printf("\n");
48
49
50
       pause();
51
52
53
        /* Encryption/Decryption (16 bits, Voice content, Reverse Channel) */
54
55
       printf("\nEncryption/Decryption (16 bits, Voice content,Reverse
56
    Channel\n\n");
57
58
        for (i = 0; i < SixOctets; i++)
59
60
           testBuf[i] = buf[i];
61
```

```
testBufTwo[i] = buf[i + 1];
1
       }
2
3
       printf(" SB Data Mask Input =");
       for (i = 0; i < SixOctets; i++)
5
          printf(" %02x", (unsigned int)testBuf[i]);
6
       printf("\n");
7
8
       Short Block Encryptor(testBuf,16,VoiceContent,testBufTwo,
9
                           ENCRYPTING, dtcScheds, ReverseChannel);
10
11
       printf("SB Data Mask Output =");
12
       for (i = 0; i < SixOctets; i++)
13
          printf(" %02x", (unsigned int)testBuf[i]);
14
       printf("\n");
15
16
       Short Block Encryptor(testBuf, 16, VoiceContent, testBufTwo,
17
                           DECRYPTING, dtcScheds, ReverseChannel);
18
19
       printf("SB Data Mask Output =");
20
       for (i = 0; i < SixOctets; i++)
21
          printf(" %02x", (unsigned int)testBuf[i]);
22
       printf("\n");
23
24
25
       /* Encryption/Decryption (2 bits, Voice content, Reverse Channel) */
26
27
       printf("\nEncryption/Decryption (2 bits, Voice content, Reverse
28
    Channel\n\n";
29
30
       for (i = 0; i < SixOctets; i++)
31
32
           testBuf[i] = buf[i];
33
           testBufTwo[i] = buf[i + 1];
34
35
36
       printf(" SB Data Mask Input =");
37
       for (i = 0; i < SixOctets; i++)
38
          printf(" %02x", (unsigned int)testBuf[i]);
39
       printf("\n");
40
41
       Short Block Encryptor(testBuf, 2, VoiceContent, testBufTwo,
42
43
                           ENCRYPTING, dtcScheds, ReverseChannel);
44
       printf("SB Data Mask Output =");
45
       for (i = 0; i < SixOctets; i++)
46
          printf(" %02x", (unsigned int)testBuf[i]);
47
       printf("\n");
48
49
       Short Block Encryptor(testBuf, 2, VoiceContent, testBufTwo,
50
                           DECRYPTING, dtcScheds, ReverseChannel);
51
52
       printf("SB Data Mask Output =");
53
       for (i = 0; i < SixOctets; i++)
54
          printf(" %02x",(unsigned int)testBuf[i]);
55
       printf("\n");
56
57
       pause();
58
59
60
       /* Encryption,47 bits, Voice content, Forward Channel */
61
```

```
1
       printf("\nEncryption,47 bits, Voice content, Forward Channel\n\n");
2
       for (i = 0; i < SixOctets; i++)
5
           testBuf[i] = buf[i];
6
           testBufTwo[i] = buf[i + 1];
8
9
       printf(" SB Data Mask Input =");
10
       for (i = 0; i < SixOctets; i++)
11
          printf(" %02x", (unsigned int)testBuf[i]);
12
       printf("\n");
13
14
       Short_Block_Encryptor(testBuf, 47, VoiceContent, testBufTwo,
15
                           ENCRYPTING, dtcScheds, ForwardChannel);
16
17
       printf("SB Data Mask Output =");
18
       for (i = 0; i < SixOctets; i++)
19
          printf(" %02x", (unsigned int)testBuf[i]);
20
       printf("\n");
21
22
23
       /* Encryption, 47 bits, Message content, Forward Channel */
24
25
       printf("\nEncryption,47 bits,Message content,Forward Channel\n\n");
26
27
       for (i = 0; i < SixOctets; i++)
28
29
           testBuf[i] = buf[i];
30
           testBufTwo[i] = buf[i + 1];
31
32
33
       printf(" SB Data Mask Input =");
34
       for (i = 0; i < SixOctets; i++)
35
          printf(" %02x",(unsigned int)testBuf[i]);
36
       printf("\n");
37
38
       Short_Block_Encryptor(testBuf, 47, MessageContent, testBufTwo,
39
                           ENCRYPTING, dtcScheds, ForwardChannel);
40
41
       printf("SB Data Mask Output =");
42
43
       for (i = 0; i < SixOctets; i++)
          printf(" %02x", (unsigned int)testBuf[i]);
44
       printf("\n");
45
46
47
48
49
       Encryption, 47 bits, Message content, Forward Channel, different entropy
50
51
       printf("\nEncryption,47 bits,Message content,Forward
52
    Channel, different entropy\n\n");
53
54
55
       for (i = 0; i < SixOctets; i++)
56
           testBuf[i] = buf[i];
57
           testBufTwo[i] = ~buf[i + 1];
58
59
60
       printf(" SB Data Mask Input =");
61
```

```
for (i = 0; i < SixOctets; i++)
1
           printf(" %02x",(unsigned int)testBuf[i]);
2
       printf("\n");
3
       Short Block Encryptor(testBuf, 47, MessageContent, testBufTwo,
6
                           ENCRYPTING, dtcScheds, ForwardChannel);
       printf("SB Data Mask Output =");
8
       for (i = 0; i < SixOctets; i++)
9
           printf(" %02x", (unsigned int)testBuf[i]);
10
       printf("\n");
11
12
       pause();
13
14
                        3.4.2.5. Vector set 6
15
16
    /* Vector Set 6 - Enhanced Message Encryption "vs6enhMsqEnc.h"
17
18
    Note: The last octets of the decrypted buffers may not match the
19
    original input buffers' last octets. This is legitimate and comprises a
20
    test to ensure that the output clean up code is working to zero out non-
21
    content bearing bits.
22
23
    * /
24
       printf("\n\nVector Set 6 - Enhanced Message Encryption\n");
25
26
       /* 48 bits */
27
28
       printf("\n48 bits\n\n");
29
30
       printf("
                    Message input =");
31
32
       for (i = 0; i < SixOctets; i++)
33
          testBuf[i] = buf[i];
34
35
       for (i = 0; i < SixOctets; i++)
36
           printf(" %02x", (unsigned int)testBuf[i]);
37
       printf("\n");
38
39
       for (i = 0; i < 4; i++)
40
          testBufTwo[i] = ~buf[i];
41
42
43
        /* Encrypting */
44
       Enhanced Message Encryption(testBuf, 48, DCCH, testBufTwo, TestMsgType,
45
                           ENCRYPTING, CAVEKey1, ForwardChannel);
46
47
48
       printf(" Encryptor output =");
49
50
51
       for (i = 0; i < SixOctets; i++)
           printf(" %02x", (unsigned int)testBuf[i]);
52
       printf("\n");
53
54
55
        /* Decrypting */
56
       Enhanced Message Encryption(testBuf, 48, DCCH, testBufTwo, TestMsgType,
57
                           DECRYPTING, CAVEKey1, ForwardChannel);
58
59
```

```
1
       printf(" Decryptor output =");
2
3
       for (i = 0; i < SixOctets; i++)
           printf(" %02x", (unsigned int)testBuf[i]);
6
       printf("\n");
8
       pause();
9
10
11
       /* 256 Octets (2047 bits) */
12
13
       printf("\n256 Octets (2047 bits)\n\n");
14
15
       printf(" Last P/O Message input =");
16
17
       for (i = 0; i < 256; i++)
18
           testBuf[i] = buf[i % EightOctets];
19
20
       for (i = 0; i < EightOctets; i++)
21
           printf(" %02x", (unsigned int)testBuf[i + 248]);
22
       printf("\n");
23
24
       for (i = 0; i < 4; i++)
25
           testBufTwo[i] = ~buf[i];
26
27
28
       /* Encrypting */
29
30
       Enhanced Message Encryption(testBuf, 2047, DCCH, testBufTwo,
31
           TestMsqType, ENCRYPTING, CAVEKey1, ForwardChannel);
32
33
34
       printf("Last P/O Encryptor output =");
35
36
       for (i = 0; i < EightOctets; i++)</pre>
37
           printf(" %02x", (unsigned int)testBuf[i + 248]);
38
       printf("\n");
39
40
41
       /* Decrypting */
42
43
       Enhanced_Message_Encryption(testBuf,2047,DCCH,testBufTwo,
44
           TestMsqType, DECRYPTING, CAVEKey1, ForwardChannel);
45
46
47
       printf("Last P/O Decryptor output =");
48
49
       for (i = 0; i < EightOctets; i++)
50
           printf(" %02x", (unsigned int)testBuf[i + 248]);
51
       printf("\n");
52
53
54
55
       pause();
56
57
       /* 44 bits */
58
59
60
       printf("\n44 bits\n\n");
61
```

```
Message input =");
                  printf("
 1
 2
                  for (i = 0; i < SixOctets; i++)
 3
                          testBuf[i] = buf[i];
                  for (i = 0; i < SixOctets; i++)
 6
                          printf(" %02x", (unsigned int)testBuf[i]);
                  printf("\n");
 8
 9
                  for (i = 0; i < 4; i++)
10
                           testBufTwo[i] = ~buf[i];
11
12
13
                   /* Encrypting */
14
                   Enhanced_Message_Encryption(testBuf,44,DCCH,testBufTwo,TestMsgType,
15
                                                                  ENCRYPTING, CAVEKey1, ForwardChannel);
16
17
18
                  printf(" Encryptor output =");
19
20
                   for (i = 0; i < SixOctets; i++)
21
                          printf(" %02x", (unsigned int)testBuf[i]);
22
                  printf("\n");
23
24
                   /* Decrypting */
25
                   {\tt Enhanced\_Message\_Encryption(testBuf,44,DCCH,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgType,testBufTwo,TestMsgT
26
27
                                                                  DECRYPTING, CAVEKey1, ForwardChannel);
28
29
                  printf(" Decryptor output =");
30
31
                   for (i = 0; i < SixOctets; i++)
32
                          printf(" %02x", (unsigned int)testBuf[i]);
33
                  printf("\n");
34
35
36
                  pause();
37
38
                  /* 48 bits, Forward Channel -> Reverse Channel */
39
40
                  printf("\n48 bits, Forward Channel -> Reverse Channel\n\n");
41
42
43
                  printf("
                                                    Message input =");
44
                   for (i = 0; i < SixOctets; i++)
45
                          testBuf[i] = buf[i];
46
47
           for (i = 0; i < SixOctets; i++)
48
                          printf(" %02x", (unsigned int)testBuf[i]);
49
                  printf("\n");
50
51
                   for (i = 0; i < 4; i++)
52
                           testBufTwo[i] = ~buf[i];
53
54
55
                   /* Encrypting */
56
                   Enhanced_Message_Encryption(testBuf,48,DCCH,testBufTwo,TestMsqType,
57
                                                                  ENCRYPTING, CAVEKey1, ReverseChannel);
58
59
60
                  printf(" Encryptor output =");
61
```

```
for (i = 0; i < SixOctets; i++)
1
          printf(" %02x", (unsigned int)testBuf[i]);
       printf("\n");
3
       /* 48 bits, DCCH -> DTC */
       printf("\n48 bits, DCCH -> DTC\n\n");
8
       printf("
                     Message input =");
9
10
       for (i = 0; i < SixOctets; i++)
11
          testBuf[i] = buf[i];
12
13
       for (i = 0; i < SixOctets; i++)
14
          printf(" %02x", (unsigned int)testBuf[i]);
15
       printf("\n");
16
17
       for (i = 0; i < 4; i++)
18
          testBufTwo[i] = ~buf[i];
19
20
       /* Encrypting */
21
       Enhanced Message Encryption(testBuf,48,DTC,testBufTwo,TestMsgType,
22
                           ENCRYPTING, CAVEKey1, ForwardChannel);
23
24
25
       printf(" Encryptor output =");
26
27
       for (i = 0; i < SixOctets; i++)
28
          printf(" %02x", (unsigned int)testBuf[i]);
29
       printf("\n");
30
31
       /* 48 bits, different RAND */
32
33
       printf("\n48 bits, different RAND\n\n");
34
35
       printf("
                   Message input =");
36
37
       for (i = 0; i < SixOctets; i++)
38
          testBuf[i] = buf[i];
39
40
       for (i = 0; i < SixOctets; i++)
41
          printf(" %02x", (unsigned int)testBuf[i]);
42
43
       printf("\n");
44
       for (i = 0; i < 4; i++)
45
          testBufTwo[i] = buf[i];
46
47
       /* Encrypting */
48
       Enhanced Message Encryption(testBuf, 48, DCCH, testBufTwo, TestMsqType,
49
                           ENCRYPTING, CAVEKey1, ForwardChannel);
50
51
       printf(" Encryptor output =");
52
53
       for (i = 0; i < SixOctets; i++)
54
          printf(" %02x", (unsigned int)testBuf[i]);
55
       printf("\n");
56
57
       /* 44 bits, different RAND */
58
59
60
       printf("\n44 bits, different RAND\n\n");
61
```

```
Message input =");
       printf("
1
       for (i = 0; i < SixOctets; i++)
3
          testBuf[i] = buf[i];
       for (i = 0; i < SixOctets; i++)
6
          printf(" %02x", (unsigned int)testBuf[i]);
       printf("\n");
8
9
       for (i = 0; i < 4; i++)
10
          testBufTwo[i] = buf[i];
11
12
       /* Encrypting */
13
       Enhanced_Message_Encryption(testBuf,44,DCCH,testBufTwo,TestMsgType,
14
                           ENCRYPTING, CAVEKey1, ForwardChannel);
15
16
       printf(" Encryptor output =");
17
18
       for (i = 0; i < SixOctets; i++)
19
          printf(" %02x", (unsigned int)testBuf[i]);
20
       printf("\n");
21
22
       /* 48 bits, different Message Type */
23
24
       printf("\n48 bits, different Message Type\n\n");
25
26
27
       printf("
                     Message input =");
28
       for (i = 0; i < SixOctets; i++)
29
          testBuf[i] = buf[i];
30
31
       for (i = 0; i < SixOctets; i++)
32
          printf(" %02x", (unsigned int)testBuf[i]);
33
       printf("\n");
34
35
       for (i = 0; i < 4; i++)
36
          testBufTwo[i] = ~buf[i];
37
38
    /* Encrypting */
39
       Enhanced_Message_Encryption(testBuf, 48, DCCH, testBufTwo,
40
                                     TestMsgType2, ENCRYPTING, CAVEKey1,
41
                                     ForwardChannel);
42
43
       printf(" Encryptor output =");
44
45
       for (i = 0; i < SixOctets; i++)
46
          printf(" %02x", (unsigned int)testBuf[i]);
47
       printf("\n");
48
49
       pause();
50
51
                        3.4.2.6. Vector set 7
52
53
    /* Vector Set 7 - Enhanced Voice Privacy "vs7enhVoicePriv.h"
54
    Note 1: The current coder standards' bit allocations as listed in
56
    TIA/EIA-136-510 are: The Number of {Class 1A bits, remaining bits, CRC
57
    bits} for 136 speech coders are: 136-410 ACELP {48, 100, 7}, 136-420
58
    VSELP {12, 147, 7}, and 136-430 US1 {81, 163, 8}.
59
```

```
1
    Note 2: The last octets of the decrypted buffers may not match the
    original input buffers' last octets. This is legitimate and comprises a
3
    test to ensure that the output clean up code is working to zero out non-
    content bearing bits.
5
6
    * /
       printf("\n\nVector Set 7 - Enhanced Voice Privacy\n");
8
9
       /* 48 Class 1A bits, 100 remaining bits */
10
11
       printf("\n48 Class 1A bits, 100 remaining bits\n\n");
12
13
       printf("1A/Rem. bits input =");
14
15
       for (i = 0; i < SixOctets; i++)
16
          testBuf[i] = buf[i];
17
18
       for (i = 0; i < SixOctets; i++)
19
          printf(" %02x", (unsigned int)testBuf[i]);
20
       printf(" /");
21
22
       for (i = 0; i < ((100 - 1) / 8) + 1; i++)
23
          testBufTwo[i] = ~buf[i % EightOctets];
24
25
       for (i = 0; i < ((100 - 1) / 8) + 1; i++)
26
          printf(" %02x", (unsigned int)testBufTwo[i]);
27
       printf("\n");
28
29
30
       /* Encrypting */
31
       Enhanced_Voice_Privacy(CoderVersionZero,testBuf,48,testBufTwo,100,
32
                           ENCRYPTING, CAVEKey1, ForwardChannel);
33
34
       printf(" Encryptor output =");
35
36
       for (i = 0; i < SixOctets; i++)
37
          printf(" %02x", (unsigned int)testBuf[i]);
38
       printf(" /");
39
40
       for (i = 0; i < ((100 - 1) / 8) + 1; i++)
41
          printf(" %02x", (unsigned int)testBufTwo[i]);
42
       printf("\n");
43
44
       /* Decrypting */
45
       Enhanced Voice Privacy(CoderVersionZero,testBuf,48,testBufTwo,100,
46
                           DECRYPTING, CAVEKey1, ForwardChannel);
47
48
       printf(" Decryptor output =");
49
50
       for (i = 0; i < SixOctets; i++)
51
          printf(" %02x",(unsigned int)testBuf[i]);
52
       printf(" /");
53
54
55
       for (i = 0; i < ((100 - 1) / 8) + 1; i++)
          printf(" %02x", (unsigned int)testBufTwo[i]);
56
       printf("\n");
57
58
       pause();
59
60
       /* 81 Class 1A bits, 163 remaining bits */
61
```

```
1
       printf("\n81 Class 1A bits, 163 remaining bits\n\n");
2
       printf("1A/Rem. bits input =");
       for (i = 0; i < ((81 - 1) / 8) + 1; i++)
6
          testBuf[i] = buf[i % EightOctets];
8
       for (i = 0; i < ((81 - 1) / 8) + 1; i++)
9
          printf(" %02x", (unsigned int)testBuf[i]);
10
       printf(" /");
11
12
       for (i = 0; i < ((163 - 1) / 8) + 1; i++)
13
          testBufTwo[i] = ~buf[i % EightOctets];
14
15
       for (i = 0; i < ((163 - 1) / 8) + 1; i++)
16
          printf(" %02x", (unsigned int)testBufTwo[i]);
17
       printf("\n");
18
19
       /* Encrypting */
20
       Enhanced Voice Privacy(CoderVersionZero,testBuf,81,testBufTwo,163,
21
                           ENCRYPTING, CAVEKey1, ForwardChannel);
22
23
       printf(" Encryptor output =");
24
25
       for (i = 0; i < ((81 - 1) / 8) + 1; i++)
26
          printf(" %02x", (unsigned int)testBuf[i]);
27
       printf(" /");
28
29
       for (i = 0; i < ((163 - 1) / 8) + 1; i++)
30
          printf(" %02x", (unsigned int)testBufTwo[i]);
31
       printf("\n");
32
33
       /* Decrypting */
34
       Enhanced Voice Privacy (CoderVersionZero, testBuf, 81, testBufTwo, 163,
35
                           DECRYPTING, CAVEKey1, ForwardChannel);
36
37
38
       printf(" Decryptor output =");
39
       for (i = 0; i < ((81 - 1) / 8) + 1; i++)
40
          printf(" %02x", (unsigned int)testBuf[i]);
41
       printf(" /");
42
43
       for (i = 0; i < ((163 - 1) / 8) + 1; i++)
44
          printf(" %02x", (unsigned int)testBufTwo[i]);
45
       printf("\n");
46
47
48
       pause();
49
       /* 12 Class 1A bits, 147 remaining bits */
50
51
       printf("\n12 Class 1A bits, 147 remaining bits\n\n");
52
53
       printf("1A/Rem. bits input =");
54
55
       for (i = 0; i < ((12 - 1) / 8) + 1; i++)
56
          testBuf[i] = buf[i % EightOctets];
57
58
       for (i = 0; i < ((12 - 1) / 8) + 1; i++)
59
          printf(" %02x", (unsigned int)testBuf[i]);
60
       printf(" /");
61
```

```
1
       for (i = 0; i < ((147 - 1) / 8) + 1; i++)
          testBufTwo[i] = ~buf[i % EightOctets];
3
       for (i = 0; i < ((147 - 1) / 8) + 1; i++)
          printf(" %02x", (unsigned int)testBufTwo[i]);
6
       printf("\n");
8
       /* Encrypting */
9
       Enhanced Voice Privacy(CoderVersionZero,testBuf,12,testBufTwo,147,
10
                           ENCRYPTING, CAVEKey1, ForwardChannel);
11
12
       printf(" Encryptor output =");
13
14
       for (i = 0; i < ((12 - 1) / 8) + 1; i++)
15
          printf(" %02x", (unsigned int)testBuf[i]);
16
       printf(" /");
17
18
       for (i = 0; i < ((147 - 1) / 8) + 1; i++)
19
          printf(" %02x", (unsigned int)testBufTwo[i]);
20
       printf("\n");
21
22
       /* Decrypting */
23
       Enhanced Voice Privacy (CoderVersionZero, testBuf, 12, testBufTwo, 147,
24
                           DECRYPTING, CAVEKey1, ForwardChannel);
25
26
       printf(" Decryptor output =");
27
28
       for (i = 0; i < ((12 - 1) / 8) + 1; i++)
29
          printf(" %02x", (unsigned int)testBuf[i]);
30
       printf(" /");
31
       for (i = 0; i < ((147 - 1) / 8) + 1; i++)
32
          printf(" %02x",(unsigned int)testBufTwo[i]);
33
       printf("\n");
34
35
       pause();
36
37
       /* Reverse Channel, 48 Class 1A bits, 100 remaining bits */
38
39
       printf("\nReverse Channel, 48 Class 1A bits, 100 remaining
40
    bits\n\n";
41
42
       printf("1A/Rem. bits input =");
43
44
       for (i = 0; i < SixOctets; i++)
45
          testBuf[i] = buf[i];
46
47
       for (i = 0; i < SixOctets; i++)
48
          printf(" %02x", (unsigned int)testBuf[i]);
49
       printf(" /");
50
51
       for (i = 0; i < ((100 - 1) / 8) + 1; i++)
52
          testBufTwo[i] = ~buf[i % EightOctets];
53
54
55
       for (i = 0; i < ((100 - 1) / 8) + 1; i++)
          printf(" %02x", (unsigned int)testBufTwo[i]);
56
       printf("\n");
57
58
       /* Encrypting */
59
       Enhanced Voice Privacy(CoderVersionZero,testBuf,48,testBufTwo,100,
60
                           ENCRYPTING, CAVEKey1, ReverseChannel);
61
```

```
1
       printf(" Encryptor output =");
2
       for (i = 0; i < SixOctets; i++)
          printf(" %02x", (unsigned int)testBuf[i]);
       printf(" /");
6
       for (i = 0; i < ((100 - 1) / 8) + 1; i++)
8
          printf(" %02x", (unsigned int)testBufTwo[i]);
9
       printf("\n");
10
11
                       3.4.2.7. Vector set 8
12
13
    /* Vector Set 8 - Enhanced Data Mask Generation "vs8enhDataMask.h" */
14
15
       printf("\nVector Set 8 - Enhanced Data Mask Generation\n\n");
16
17
       Enhanced Data Mask(testBuf, 0x87654321, SixOctets, CAVEKey1);
18
19
       printf("Enhanced Data Mask Output =");
20
       for (i = 0; i < SixOctets; i++)
21
          printf(" %02x", (unsigned int)testBuf[i]);
22
       printf("\n");
23
24
       Enhanced Data Mask(testBuf, 0x87654321, 3, CAVEKey1);
25
26
       printf(" Output, with short Mask =");
27
       for (i = 0; i < 3; i++)
28
          printf(" %02x", (unsigned int)testBuf[i]);
29
       printf("\n\n");
30
31
       pause();
32
                 3.4.3. Test Program Input and Output
33
34
    Vector Set 1 - DTC Key Generation and SCEMA
35
36
                DTC CMEA key = a0 7b 1c d1 02 75 69 14
    DTC scemaKey (CaveKey1) = 5d ed ad 53 5b 4a b9 fc
37
                        sync = 3d 00 a2 00
38
                        Input = b6 2d a2 44 fe 9b
39
40
            DTC SCEMA Output = 63 f0 21 7a 3c 97
41
42
    Vector Set 2 - DCCH Key Generation and SCEMA
43
               DCCH CMEA key = f0 06 a8 5a 05 cd b3 2a
               DCCH CMEA key = f0 06 a8 5a 05 cd b3 2a
44
    DCCH scemaKey (CaveKey1) = b6 df 9a d0 6e 5a 3d 14
45
    DCCH scemaKey (CaveKey1) = b6 df 9a d0 6e 5a 3d 14
46
47
                        sync = ff 00 ff 00
                       Input = b6 2d a2 44 fe 9b
48
          DCCH SCEMA Output = 4c 3d 77 13 e9 a0
49
50
```

```
Vector Set 3 - SCEMA KSG
    Voice content, Reverse Channel, 3-octet input, 8-octet output
3
                Input = b6 2d a2
    SCEMA KSG Output = f4 bc 1e 9b 27 a1 54 fa
4
    Voice content, Reverse Channel, 6-octet input, 6-octet output
6
                Input = b6 2d a2 44 fe 9b
7
    SCEMA KSG Output = 26 08 0c fa d2 7d
8
    Voice content, Reverse Channel, 6-octet input, 3-octet requested output,
10
    6-octets delivered
11
12
                Input = b6 2d a2 44 fe 9b
    SCEMA KSG Output = 26 08 0c fa d2 7d
13
14
    Message content, Reverse Channel, 6-octet input, 6-octet output
15
                Input = b6 2d a2 44 fe 9b
16
    SCEMA KSG Output = df 39 6c 92 c8 63
17
18
    Message content, Forward Channel, 6-octet input, 6-octet output
19
                Input = b6 2d a2 44 fe 9b
20
    SCEMA KSG Output = 8c a4 9a f5 54 53
21
22
    Vector Set 4 - Long Block Encryptor
23
    Encryption/Decryption (Voice content, Reverse Channel)
24
25
                           Input = b6 2d a2 44 fe 9b
    Long Block Encryptor Output = 59 fe 84 59 ec 18
26
    Long Block Decryptor Output = b6 2d a2 44 fe 9b
27
28
    Encryption (Message Content, Reverse Channel)
29
30
                           Input = b6 2d a2 44 fe 9b
    Long Block Encryptor Output = 53 7e d4 c6 37 98
31
32
    Encryption (Voice Content, Forward Channel)
33
                           Input = b6 2d a2 44 fe 9b
34
    Long Block Encryptor Output = bd 5e 36 a5 8c 07
35
```

```
1
    Vector Set 5 - Short Block Encryptor
    Encryption/Decryption (47 bits, Voice content, Reverse Channel)
    SB Data Mask Input = b6 2d a2 44 fe 9b
    SB Encryptor Output = af f8 41 7e 5d f2
    SB Decryptor Output = b6 2d a2 44 fe 9a
6
    Encryption/Decryption (17 bits, Voice content, Reverse Channel
8
     SB Data Mask Input = b6 2d a2 44 fe 9b
9
    SB Encryptor Output = b7 ed 80 00 00 00
10
    SB Decryptor Output = b6 2d 80 00 00 00
11
12
13
    Encryption/Decryption (16 bits, Voice content, Reverse Channel
14
    SB Data Mask Input = b6 2d a2 44 fe 9b
    SB Encryptor Output = 9b a8 00 00 00 00
15
    SB Decryptor Output = b6 2d 00 00 00 00
16
17
    Encryption/Decryption (2 bits, Voice content, Reverse Channel
18
    SB Data Mask Input = b6 2d a2 44 fe 9b
19
    SB Encryptor Output = 00 00 00 00 00 00
20
21
    SB Decryptor Output = 80 00 00 00 00 00
22
    Encryption, 47 bits, Voice content, Forward Channel
23
     SB Data Mask Input = b6 2d a2 44 fe 9b
24
    SB Encryptor Output = 9e df 05 a8 43 34
25
26
    Encryption, 47 bits, Message content, Forward Channel
27
     SB Data Mask Input = b6 2d a2 44 fe 9b
28
    SB Encryptor Output = 4f 89 f7 09 29 a8
29
30
    Encryption, 47 bits, Message content, Forward Channel, different entropy
31
```

SB Data Mask Input = b6 2d a2 44 fe 9b SB Encryptor Output = c8 fe da 7d 87 da

48 bits Message input = b6 2d a2 44 fe 9b 3 Encryptor output = 87 ce 86 a0 f1 86 Decryptor output = b6 2d a2 44 fe 9b 256 Octets (2047 bits) Last P/O Message input = b6 2d a2 44 fe 9b 23 ab 8 Last P/O Encryptor output = 2b 52 46 a6 da 82 f2 f0 9 Last P/O Decryptor output = b6 2d a2 44 fe 9b 23 aa 10 44 bits 12 13 Message input = b6 2d a2 44 fe 9b Encryptor output = b4 5b 16 d1 c2 10 Decryptor output = b6 2d a2 44 fe 90 16 48 bits, Forward Channel -> Reverse Channel Message input = b6 2d a2 44 fe 9b 18 Encryptor output = 28 09 3e fe 49 06 19 20 48 bits, DCCH -> DTC 21 22 Message input = b6 2d a2 44 fe 9b Encryptor output = 28 a4 ed a0 68 0a 23 48 bits, different RAND 25 Message input = b6 2d a2 44 fe 9b 26 Encryptor output = 3c cf 9e 23 a5 7c 27 44 bits, different RAND Message input = b6 2d a2 44 fe 9b Encryptor output = a7 03 f3 42 2b 10 32 48 bits, different Message Type 33 Message input = b6 2d a2 44 fe 9b 34 Encryptor output = dc 27 53 82 d5 77 35

36

Vector Set 6 - Enhanced Message Encryption

Vector Set 7 - Enhanced Voice Privacy

2 48 Class 1A bits, 100 remaining bits

```
3 1A/Rem. bits input = b6 2d a2 44 fe 9b / 49 d2 5d bb 01 64 dc 54 49 d2 5d bb 01 5d bb 01 5d bb 01 5e 36 a5 8c 07 / 87 58 05 c7 38 37 0f 68 e2 3f 6 d4 5c 30 7 Decryptor output = b6 2d a2 44 fe 9b / 49 d2 5d bb 01 64 dc 54 49 d2 5d bb 00
```

9

10 81 Class 1A bits, 163 remaining bits

```
11 1A/Rem. bits input = b6 2d a2 44 fe 9b 23 ab b6 2d a2 / 49 d2 5d bb 01
12 64 dc 54 49 d2 5d bb 01 64 dc 54 49 d2 5d bb 01
13 Encryptor output = 5b 68 57 98 42 83 81 92 b2 1f 80 / e8 52 c6 f6 60
14 39 16 a0 80 c7 b0 59 fb 5c 6e 23 91 08 bc d2 a0
15 Decryptor output = b6 2d a2 44 fe 9b 23 ab b6 2d 80 / 49 d2 5d bb 01
16 64 dc 54 49 d2 5d bb 01 64 dc 54 49 d2 5d bb 00
```

17

18 12 Class 1A bits, 147 remaining bits

25

26 Reverse Channel, 48 Class 1A bits, 100 remaining bits

31

32

Vector Set 8 - Enhanced Data Mask Generation

```
Enhanced Data Mask Output = 45 b0 15 31 d6 e0 Output, with short mask = 45 b0 15
```