TR45.0.A

Common Cryptographic Algorithms, Revision B

June 21, 1995

Information disclosed in this document may be subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive)

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 State as specified in International Traffic in Arms Regulations (ITAR), Title 22 CFR parts 120 through 130 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
2001 Pennsylvania Avenue, N.W., Suite 800
Washington, D.C. 20006-1813

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
B	04-26-95	Add procedures for wireless residential extension authentication

No text.

Table of Contents

No text.

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130

1. Introduction

This document describes detailed cryptographic procedures for cellular system applications. These procedures are used to perform the security services of mobile station authentication, subscriber message encryption, and encryption key and subscriber voice privacy key generation within cellular equipment.

This document is organized as follows:

§2 describes the Cellular Authentication, Voice Privacy and Encryption (CAVE) algorithm used for authentication of mobile subscriber equipment and for generation of cryptovariables to be used in other procedures.

§2.1 describes the procedure to verify the manual entry of the subscriber authentication key (A-key).

§2.2 describes the generation of intermediate subscriber cryptovariables, Shared Secret Data (SSD), from the unique and private subscriber A-key.

§2.3 describes the procedure to calculate an authentication signature used by cellular base station equipment for verifying the authenticity of a mobile station.

§2.4 describes the procedures used for generating cryptographic keys. These keys include the Voice Privacy Mask (VPM) and the Cellular Message Encryption Algorithm (CMEA) key. The VPM is used to provide forward link and reverse link voice confidentiality over the air interface. The CMEA key is used with the CMEA algorithm for protection of digital data exchanged between the mobile station and the base station.

§2.5 describes the Cellular Message Encryption Algorithm (CMEA), used for enciphering and deciphering subscriber data exchanged between the mobile station and the base station.

§2.6 describes the procedures for key and authentication signature generation for wireless residential extension applications.

§2.7 describes the ORYX algorithm and procedures for key and mask generation for encryption and decryption in cellular data services.

§3 provides test data (vectors) that may be employed to verify the correct operation of the cryptographic algorithms described in this 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 byte (LFSR A7-A0) and the least significant bits into its rightmost byte (LFSR D7-D0).

This document uses ANSI C language programming syntax to specify the behavior of the cryptographic algorithms. 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

AAV

Authentication Algorithm Version, an 8-bit constant equal to hexadecimal 0xC7, used in the algorithm. Use of different values for this constant in some future version would allow other "versions" or "flavors" of the basic CAVE algorithm.

ACRE

Authorization and Call Routing Equipment. A network device which authorizes the Personal Base and provides automatic call routing.

ACRE_PHONE_NUMBER

A 24-bit pattern comprised of the last 6 digits of the ACRE's directory number.

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 cellular system. It is entered once from the keypad of the mobile station 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.

AND

Bitwise logical AND function.

Boolean Describes a quantity whose value is either TRUE or FALSE.

CAVE Cellular Authentication and Voice Encryption algorithm.

CaveTable A lookup table consisting of 256 8-bit quantities. The table, partitioned into table0 and table1, is used in the CAVE algorithm.

CMEA Cellular Message Encryption Algorithm.

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.

DataKey A 32-bit cryptographic key used for generation of masks for encryption and decryption in cellular data services.

Directory Number The telephone network address.

ESN The 32-bit electronic serial number of the mobile station.

Internal Stored Data Stored data that is defined locally within the cryptographic procedures and is not accessible for examination or use outside those procedures.

Iteration Multi-round execution of the CAVE algorithm. All applications of CAVE throughout this document use either four or eight rounds per iteration.

k0,k1...k7 Eight 8-bit registers whose contents constitute the CMEA key.

LFSR A 32-bit Linear Feedback Shift Register, which is composed of four 8-bit registers.

LFSR_A The A register, a synonym for bits 31-24 of the LFSR.

LFSR_B The B register, a synonym for bits 23-16 of the LFSR.

LFSR_C The C register, a synonym for bits 15-8 of the LFSR.

LFSR_D The D register, a synonym for bits 7-0 of the LFSR.

LFSR-Cycle An LFSR-cycle consists of the following steps:

1. 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.
2. Perform a linked 1-bit right shift on the 32-bit LFSR, and discard the D0 bit which has been shifted out.
3. Use the previously computed and stored value of bit A7 from the first of these three statements.

LSB Least Significant Bit.

MSB Most Significant Bit.

OR Bitwise logical inclusive OR function.

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.

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.

PB Personal Base. A fixed device which provides cordless like service to a mobile station.

PBID
Personal Base Identification Code.

RAND_ACRE
A 32-bit random number which is generated by the PB.

RAND_PB
A 32-bit random number which is generated by the ACRE.

RAND_WIKEY
A 56-bit random number which is generated by the ACRE.

RAND_WRE
A 19-bit random number which is generated by the PB.

Round
A round is one individual execution of the CAVE algorithm.

R00-R15
Sixteen separate 8-bit mixing registers used in the CAVE algorithm. Also called register[0 through 15].

SSD
SSD is an abbreviation for Shared Secret Data. It consists of two quantities, SSD_A and SSD_B.

SSD_A
A 64-bit binary quantity in the semi-permanent memory of the mobile station and also known to the serving MSC. It is used in the computation of the authentication response.

SSD_A_NEW
The revised 64-bit quantity held separately from SSD_A, generated as a result of the SSD generation process.

SSD_B
A 64-bit binary quantity in the semi-permanent memory of the mobile station and also known to the serving MSC. It is used in the computation of the CMEA key, VPM and DataKey.

SSD_B_NEW
The revised 64-bit quantity held separately from SSD_B, generated as a result of the SSD generation process.

table0
The low-order four bits of the 256-byte lookup table used in the CAVE algorithm. Computed as `CaveTable[] AND 0x0F`.

table1
The high-order four bits of the 256-byte lookup table used in the CAVE algorithm. Computed as `CaveTable[] AND 0xF0`.

VPM
Voice Privacy Mask. This name describes a 520-bit entity that may be used for voice privacy functions as specified in cellular system standards.

WIKEY
Wireline Interface key. A 64-bit pattern stored in the PB and the ACRE (in semi-permanent memory).

WIKEY_NEW
A 64-bit pattern stored in the PB and the ACRE. It contains the value of an updated WIKEY.

WRE_KEY
Wireless Residential Extension key. A 64-bit pattern stored in the PB and the MS (in semi-permanent memory).

XOR
Bitwise logical exclusive OR.

2. Procedures

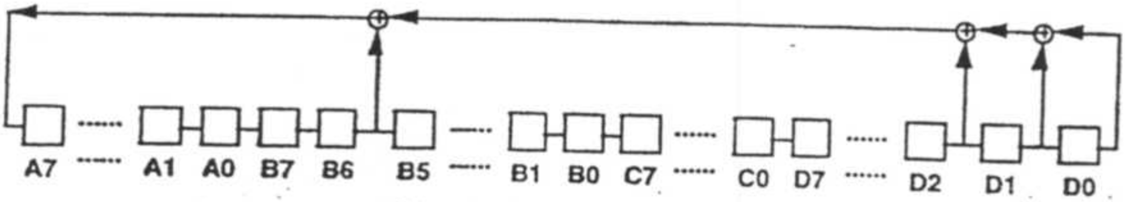
CAVE is a software-compatible non-linear mixing function shown in Exhibit 2-1. Its primary components are a 32-bit linear-feedback shift register (LFSR), sixteen 8-bit mixing registers, and a 256-entry lookup table. The table is organized as two (256 x 4 bit) tables. The 256-byte table is listed in Exhibit 2-3. 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. 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 2-3. 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 §2.1 through §2.4 of this document.

Exhibit 2-1 CAVE Elements



A7(input) = B6 XOR D2 XOR D1 XOR D0

Mixing Registers:

R00	R01	R02	R03	R04	R05	R06	R07	R08	R09	R10	R11	R12	R13	R14	R15

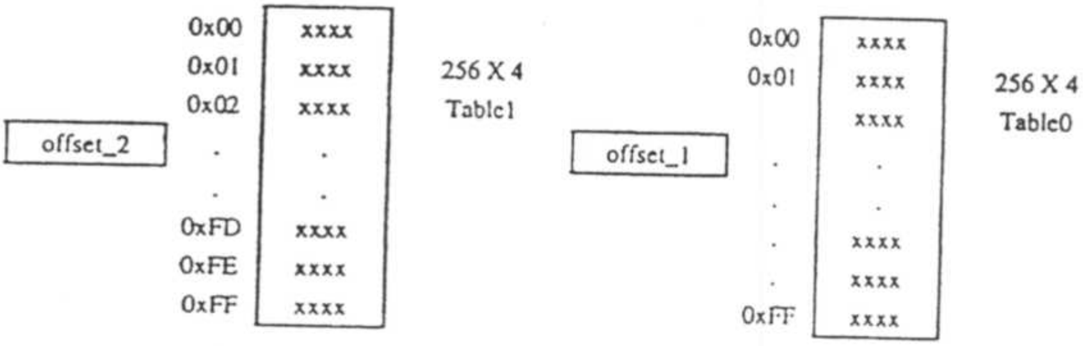


Exhibit 2-2 CAVE Algorithm

```
/* header for CAVE and related procedures */

void CAVE(const int number_of_rounds,
          int *offset_1,
          int *offset_2);

void A_Key_Checksum(const char A_KEY_DIGITS[20],
                    char A_KEY_CHECKSUM[6]);

int A_Key_Verify(const char A_KEY_DIGITS[26]);

void SSD_Generation(const unsigned char RANDSSD[7]);

unsigned long Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                             const unsigned char AUTH_DATA[3],
                             const unsigned char *SSD_AUTH,
                             const int SAVE_REGISTERS);

void Key_VPM_Generation(void);

void CMEA(unsigned char *msg_buf, const int byte_count);

void WIKEY_Generation(const unsigned char MANUFACT_KEY[16],
                      const unsigned char PBID[4]);

void WIKEY_Update(const unsigned char RANDWIKEY[7],
                  const unsigned char PBID[4]);

unsigned long WI_Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                                const unsigned char PBID[4],
                                const unsigned char ACRE_PHONE_NUMBER[3]);

unsigned long WRE_Auth_Signature(const unsigned char RAND_WRE[3],
                                 const unsigned char PBID[4],
                                 const unsigned char ESN[4]);

#define LOMASK        0x0F
#define HIMASK        0xF0
#define TRUE          1
#define FALSE         0

/* authentication algorithm version (fixed) */

unsigned char          AAV = { 0xc7 };

/* external input data */

unsigned char     ESN[4];

/* saved outputs */

unsigned char     LFSR[4];

#define LFSR_A LFSR[0]
#define LFSR_B LFSR[1]
#define LFSR_C LFSR[2]
```

```
#define LFSR_D LFSR[3]

unsigned char      Register[16];

unsigned char      A_key[8];
unsigned char      SSD_A_NEW[8], SSD_A[8];
unsigned char      SSD_B_NEW[8], SSD_B[8];
unsigned char      cmeakey[8];
unsigned char      VPM[65];

unsigned char      SAVED_LFSR[4];
int                SAVED_OFFSET_1;
int                SAVED_OFFSET_2;
unsigned char      SAVED_RAND[4];
unsigned char      SAVED_DATA[3];
unsigned char      SAVED_ESN[4];

unsigned char      WIKEY[8];
unsigned char      WIKEY_NEW[8];
unsigned char      WRE_KEY[8];


/* table0 is the 4 lsbs of the array,
   table1 is the 4 msbs of the array */

```

```
unsigned char     CaveTable[256] =

    {  0xd9, 0x23, 0x5f, 0xe6, 0xca, 0x68, 0x97, 0xb0,
       0x7b, 0xf2, 0x0c, 0x34, 0x11, 0xa5, 0x8d, 0x4e,
       0x0a, 0x46, 0x77, 0x8d, 0x10, 0x9f, 0x5e, 0x62,
       0xf1, 0x34, 0xec, 0xa5, 0xc9, 0xb3, 0xd8, 0x2b,
       0x59, 0x47, 0xe3, 0xd2, 0xff, 0xae, 0x64, 0xca,
       0x15, 0x8b, 0x7d, 0x38, 0x21, 0xbc, 0x96, 0x00,
       0x49, 0x56, 0x23, 0x15, 0x97, 0xe4, 0xcb, 0x6f,
       0xf2, 0x70, 0x3c, 0x88, 0xba, 0xd1, 0x0d, 0xae,
       0xe2, 0x38, 0xba, 0x44, 0x9f, 0x83, 0x5d, 0x1c,
       0xde, 0xab, 0xc7, 0x65, 0xf1, 0x76, 0x09, 0x20,
       0x86, 0xbd, 0x0a, 0xf1, 0x3c, 0xa7, 0x29, 0x93,
       0xcb, 0x45, 0x5f, 0xe8, 0x10, 0x74, 0x62, 0xde,
       0xb8, 0x77, 0x80, 0xd1, 0x12, 0x26, 0xac, 0x6d,
       0xe9, 0xcf, 0xf3, 0x54, 0x3a, 0x0b, 0x95, 0x4e,
       0xb1, 0x30, 0xa4, 0x96, 0xf8, 0x57, 0x49, 0x8e,
       0x05, 0x1f, 0x62, 0x7c, 0xc3, 0x2b, 0xda, 0xed,
       0xbb, 0x86, 0x0d, 0x7a, 0x97, 0x13, 0x6c, 0x4e,
       0x51, 0x30, 0xe5, 0xf2, 0x2f, 0xd8, 0xc4, 0xa9,
       0x91, 0x76, 0xf0, 0x17, 0x43, 0x38, 0x29, 0x84,
       0xa2, 0xdb, 0xef, 0x65, 0x5e, 0xca, 0x0d, 0xbc,
       0xe7, 0xfa, 0xd8, 0x81, 0x6f, 0x00, 0x14, 0x42,
       0x25, 0x7c, 0x5d, 0xc9, 0x9e, 0xb6, 0x33, 0xab,
       0x5a, 0x6f, 0x9b, 0xd9, 0xfe, 0x71, 0x44, 0xc5,
       0x37, 0xa2, 0x88, 0x2d, 0x00, 0xb6, 0x13, 0xec,
       0x4e, 0x96, 0xa8, 0x5a, 0xb5, 0xd7, 0xc3, 0x8d,
       0x3f, 0xf2, 0xec, 0x04, 0x60, 0x71, 0x1b, 0x29,
       0x04, 0x79, 0xe3, 0xc7, 0x1b, 0x66, 0x81, 0x4a,
       0x25, 0x9d, 0xdc, 0x5f, 0x3e, 0xb0, 0xf8, 0xa2,
       0x91, 0x34, 0xf6, 0x5c, 0x67, 0x89, 0x73, 0x05,
       0x22, 0xaa, 0xcb, 0xee, 0xbf, 0x18, 0xd0, 0x4d,
       0xf5, 0x36, 0xae, 0x01, 0x2f, 0x94, 0xc3, 0x49,
       0x8b, 0xbd, 0x58, 0x12, 0xe0, 0x77, 0x6c, 0xda  };

/* end of CAVE header */
```

```
/*******************************************************************
*****************************************************************/
static unsigned char bit_val(const unsigned char byte, const int bit)
{
   return((byte << (7 - bit)) & 0x80);
}

static void LFSR_cycle(void)
{
   unsigned char temp;
   int i;

   temp  = bit_val(LFSR_B,6);
   temp ^= bit_val(LFSR_D,2);
   temp ^= bit_val(LFSR_D,1);
   temp ^= bit_val(LFSR_D,0);

   /* Shift right LFSR, Discard LFSR_D[0] bit */

   for (i = 3; i > 0; i--)
   {
      LFSR[i] >>= 1;
      if (LFSR[i-1] & 0x01)
         LFSR[i] |= 0x80;
   }
   LFSR[0] >>= 1;

   LFSR_A |= temp;
}

static void Rotate_right_registers(void)
{
   unsigned int temp_reg;
   int i;

   temp_reg = Register[15]; /* save lsb */

   for (i = 15; i > 0; i--)
   {
      Register[i] >>= 1;
      if (Register[i-1] & 0x01)
         Register[i] |= 0x80;
   }

   Register[0] >>= 1;
   if (temp_reg & 0x01)
      Register[0] |= 0x80;
}
```

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

```
void CAVE(const int number_of_rounds,
          int *offset_1,
          int *offset_2)
{
   unsigned char      temp_reg0;
   unsigned char      lowNibble;
   unsigned char      hiNibble;
   unsigned char      temp;
   int                round_index;
   int                R_index;
   int                fail_count;
   unsigned char      T[16];

   for (round_index = number_of_rounds - 1;
        round_index >= 0;
        round_index--)
   {
      /* save R0 for reuse later */
      temp_reg0 = Register[0];

      for (R_index = 0; R_index < 16; R_index++)
      {
         fail_count = 0;
         while(1)
         {
            *offset_1 += (LFSR_A ^ Register[R_index]);
            /* will overflow; mask to prevent */
            *offset_1 &= 0xff;
            lowNibble = CaveTable[*offset_1] & LOMASK;
            if (lowNibble == (Register[R_index] & LOMASK))
            {
               LFSR_cycle();
               fail_count++;
               if (fail_count == 32)
               {
                  LFSR_D++; /* no carry to LFSR_C */
                  break;
               }
            }
            else break;
         }
         fail_count = 0;
         while(1)
         {
            *offset_2 += (LFSR_B ^ Register[R_index]);
            /* will overflow; mask to prevent */
            *offset_2 &= 0xff;
            hiNibble = CaveTable[*offset_2] & HIMASK;
            if (hiNibble == (Register[R_index] & HIMASK))
            {
               LFSR_cycle();
               fail_count++;
               if (fail_count == 32)
               {
                  LFSR_D++; /* no carry to LFSR_C */
                  break;
               }
            }
            else
               break;
         }

         temp = lowNibble | hiNibble;
         if (R_index == 15)
            Register[R_index] = temp_reg0 ^ temp;
         else
            Register[R_index] = Register[R_index+1] ^ temp;

         LFSR_cycle();
      }

      Rotate_right_registers();

      /* shuffle the mixing registers */
      for (R_index = 0; R_index < 16; R_index++)
      {
         temp = CaveTable[16*round_index + R_index] & LOMASK;
         T[temp] = Register[R_index];
      }
      for (R_index = 0; R_index < 16; R_index++)
         Register[R_index] = T[R_index];
   }
}
```

Exhibit 2-3 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	B	C	D	E	F
0	D9	23	5F	E6	CA	68	97	B0	7B	F2	0C	34	11	A5	8D	4E
1	0A	46	77	8D	10	9F	5E	62	F1	34	EC	A5	C9	B3	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	CB	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	CB	45	5F	E8	10	74	62	DE
6	B8	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	B6	33	AB
B	5A	6F	9B	D9	FE	71	44	C5	37	A2	88	2D	00	B6	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	B0	F8	A2
E	91	34	F6	5C	67	89	73	05	22	AA	CB	EE	BF	18	D0	4D
F	F5	36	AE	01	2F	94	C3	49	8B	BD	58	12	E0	77	6C	DA

2.1. Authentication Key (A-Key) Procedures

2.1.1. A-Key Checksum Calculation

Procedure name:	
A_Key_Checksum	
Inputs from calling process:	
A_KEY_DIGITS	20 decimal digits
ESN	32 bits
Inputs from internal stored data:	
AAV	8 bits
Outputs to calling process:	
A_KEY_CHECKSUM	6 decimal digits
Outputs to internal stored data:	
	None.

This procedure computes the checksum for an A-key to be entered into a mobile station. In a case where the number of digits to be entered is less than 20, the leading most significant digits will be set equal to zero.

The generation of the A-key is the responsibility of the service provider. A-keys should be chosen and managed using procedures that minimize the likelihood of compromise.

The checksum provides a check for the accuracy of the A-Key when entered into a mobile station. The 20 A-Key digits are converted into a 64-bit representation to serve as an input to CAVE, along with the mobile station's ESN. CAVE is then run in the same manner as for the Auth_Signature procedure, and its 18-bit response is the A-Key checksum. The checksum is returned as 6 decimal digits for entry into the mobile station.

The first decimal digit of the A-Key to be entered is considered to be the most significant of the 20 decimal digits, followed in succession by the other nineteen. A decimal to binary conversion process converts the digit sequence into its equivalent mod-2 representation. For example, the 20 digits

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0

have a hexadecimal equivalent of

A B 5 4 A 9 8 C E B 1 F 0 A D 2.

CAVE will be initialized as shown in Exhibit 2-4. 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

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

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. 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 2-4 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 2-5 A-key Checksum

```
/* A_Key_Checksum has the same header as CAVE (see Exhibit 2-2) */

static void mul10(unsigned char i64[8],
                  unsigned int carry)
{
   int i;
   unsigned int temp;

   for (i = 7; i >= 0; i--)
   {
      temp = ((unsigned int)(i64[i]) * 10) + carry;
      i64[i] = temp & 0xFF;
      carry =  temp >> 8;
   }
}

static unsigned long Calc_Checksum(const unsigned char A_key[8])
{
   int i,offset_1,offset_2;
   unsigned long A_key_checksum;


   /* see if 32 MSB are zero */

   if ((A_key[0] | A_key[1] | A_key[2] | A_key[3]) != 0)
   {
      /* put 32 MSB into LFSR */
      for (i = 0; i < 4; i++)
         LFSR[i] = A_key[i];
   }
   else
   {
      /* put ESN into LFSR */
      for (i = 0; i < 4; i++)
         LFSR[i] = ESN[i];
   }

   /* put A_key into r0-r7 */

   for (i = 0; i < 8; i++)
      Register[i] = A_key[i];

   Register[8] = AAV;

   /* put ls 24 bits of A_key into r9-r11 */

   for (i = 9; i < 12; i++)
      Register[i] = A_key[5+i-9];

   /* put ESN into r12-r15 */
   for (i = 12; i < 16; i++)
      Register[i] = ESN[i-12];

   offset_1 = offset_2 = 128;

   CAVE(8, &offset_1, &offset_2);
```

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

```

   A_key_checksum =
             ( ((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);
}

/* A_KEY_DIGITS contains the ASCII digits in the order to be entered */

void A_Key_Checksum(const char A_KEY_DIGITS[20],
                    char A_KEY_CHECKSUM[6])
{
   int i,offset_1,offset_2;
   unsigned char temp_A_key[8];
   unsigned long A_key_checksum;

   /* convert digits to 64-bit representation in temp_A_key */

   for (i = 0; i < 8; i++)
      temp_A_key[i] = 0;

   for (i = 0; i < 20; i++)
   {
      mul10(temp_A_key,(unsigned int)(A_KEY_DIGITS[i] - '0'));
   }

   A_key_checksum = Calc_Checksum(temp_A_key);

   /* convert checksum to decimal digits */

   for (i = 0; i < 6; i++)
   {
      A_KEY_CHECKSUM[5-i] = '0' + (A_key_checksum % 10);
      A_key_checksum /= 10;
   }
}
```

2.1.2. A-Key Verification

Procedure name:	
A_Key_Verify	
Inputs from calling process:	
A_KEY_DIGITS	from 6 to 26 decimal digits
ESN	32 bits
Inputs from internal stored data:	
AAV	8 bits
Outputs to calling process:	
A_KEY_VERIFIED	Boolean
Outputs to internal stored data:	
A-key	64 bits
SSD_A	64 bits (set to zero)
SSD_B	64 bits (set to zero)

This procedure verifies the A-key entered into a mobile station.

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. See "User Interface for Authentication Key Entry," TSB-50, for details of A-key entry into the mobile station.

While A-key digits are being entered, 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. The computed A-Key checksum is compared to the binary equivalent of the last six entered digits. A match will cause the 64-bit pattern determined by the first 20 digits of the entry value to be written to the subscriber's semi-permanent memory as the A-key. The return value A_KEY_VERIFIED will be set to TRUE. Furthermore, the SSD_A and the SSD_B will be set to zero. In the case of a mismatch, A_KEY_VERIFIED is set to FALSE, and no internal data is updated.

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. A decimal to binary conversion process converts both digit sequences into their equivalent mod-2 representation. For example, the 26 digits

12345678901234567890, 131136

has a hexadecimal equivalent of

AB54A98CEB1F0AD2, 20040.

The computed 18-bit A-Key checksum will be compared to the binary equivalent of the last six entered digits. A match will enable semi-permanent storage. A mismatch can initiate corrective action.

Exhibit 2-6 A-key Verification

```
/* A_Key_Verify has the same header as CAVE (see Exhibit 2-2) */

/* A_KEY_DIGITS contains the ASCII digits in the order entered */

int A_Key_Verify(const char A_KEY_DIGITS[26])
{
   int i,offset_1,offset_2;
   unsigned char temp_A_key[8];
   unsigned long entered_checksum;

   /* convert first 20 digits to 64-bit representation in temp_A_key */

   for (i = 0; i < 8; i++)
      temp_A_key[i] = 0;

   for (i = 0; i < 20; i++)
   {
      mul10(temp_A_key,(unsigned int)(A_KEY_DIGITS[i] - '0'));
   }

   /* convert last 6 digits to entered checksum */

   entered_checksum = 0;
   for (i = 20; i < 26; i++)
   {
      entered_checksum = (entered_checksum * 10)
         + (A_KEY_DIGITS[i] - '0');
   }

   if(Calc_Checksum(temp_A_key) == entered_checksum)
   {
      for (i = 0; i < 8; i++)
      {
         A_key[i] = temp_A_key[i];
         SSD_A[i] = SSD_B[i] = 0;
      }
      return TRUE;
   }
   else
   {
      return FALSE;
   }
}
```

2.2. SSD Generation and Update

2.2.1. SSD Generation Procedure

Procedure name:

SSD_Generation

Inputs from calling process:

RANDSSD	56 bits
ESN	32 bits

Inputs from internal stored data:

AAV	8 bits
A-key	64 bits

Outputs to calling process:

None.

Outputs to internal stored data:

SSD_A_NEW	64 bits
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 (§2.2.2) is invoked. Exhibit 2-7 shows the process graphically. Exhibit 2-8 indicates the operations in ANSI C.

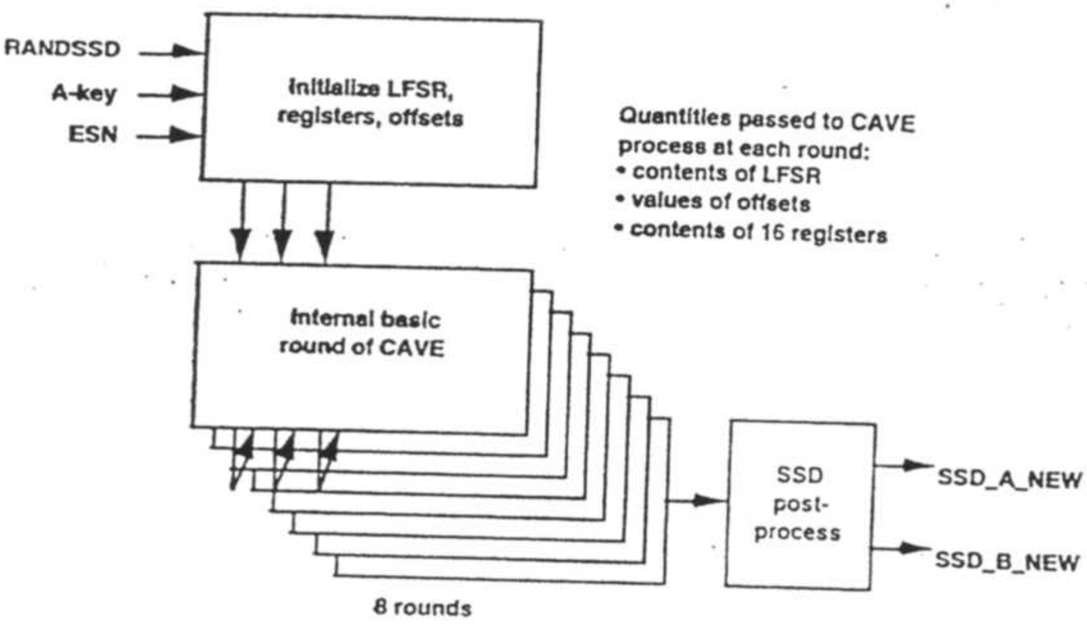
The input variables for this procedure are: RANDSSD (56 bits), Authentication Algorithm Version (8 bits), ESN (32 bits), and A-key (64 bits). CAVE will be initialized as follows. First, the LFSR will be loaded with the 32 least significant bits of RANDSSD XOR'd with the 32 most significant bits of A-key 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. When this is complete, registers R00 through R07 will become SSD_A_NEW and Registers R08 through R15 will become SSD_B_NEW.

1

Exhibit 2-7 Generation of SSD_A_NEW and SSD_B_NEW


RANDSSD
A-key
ESN
Initialize LFSR, registers, offsets
Quantities passed to CAVE process at each round:
• contents of LFSR
• values of offsets
• contents of 16 registers
Internal basic round of CAVE
8 rounds
SSD post-process
SSD_A_NEW
SSD_B_NEW

2

Exhibit 2-8 SSD Generation

```
/* SSD_Generation has the same header as CAVE (see Exhibit 2-2) */

void SSD_Generation(const unsigned char RANDSSD[7])
{
   int i,offset_1,offset_2;

   for (i = 0; i < 4; i++)
   {
      LFSR[i] = RANDSSD[i+3] ^ A_key[i] ^ A_key[i+4];
   }

   if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = RANDSSD[i+3];
   }

   for (i = 0; i < 8; i++)
      Register[i] = A_key[i];

   Register[8] = AAV;

   for (i = 9; i < 12; i++)
      Register[i] = RANDSSD[i-9];

   for (i = 12; i < 16; i++)
      Register[i] = ESN[i-12];

   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);

   for (i = 0; i < 8; i++)
   {
      SSD_A_NEW[i] = Register[i];
      SSD_B_NEW[i] = Register[i+8];
   }
}
```

2.2.2. SSD Update Procedure

Procedure name:

SSD_Update

Inputs from calling process:

None.

Inputs from internal stored data:

SSD_A_NEW	64 bits
SSD_B_NEW	64 bits

Outputs to calling process:

None.

Outputs to internal stored data:

SSD_A	64 bits
SSD_B	64 bits

This procedure copies the values SSD_A_NEW and SSD_B_NEW into the stored SSD_A and SSD_B.

The values SSD_A_NEW and SSD_B_NEW calculated by the SSD_Generation procedure (§2.2.1) 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.

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 2-9 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 2-10 shows the process in graphical form, while ANSI C for the process is given in Exhibit 2-11.

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.

2.3. Authentication Signature Calculation Procedure

Procedure name:

Auth_Signature

Inputs from calling process:

RAND_CHALLENGE	32 bits
ESN	32 bits
AUTH_DATA	24 bits
SSD_AUTH	64 bits
SAVE_REGISTERS	Boolean

Inputs from internal stored data:

AAV	8 bits

Outputs to calling process:

AUTH_SIGNATURE	18 bits

Outputs to internal stored data:

SAVED_LFSR	32 bits
SAVED_OFFSET_1	8 bits
SAVED_OFFSET_2	8 bits
SAVED_RAND	32 bits
SAVED_DATA	24 bits
SAVED_ESN	32 bits

This procedure is used to calculate 18-bit signatures used for verifying the authenticity of messages used to request cellular system services, and for verifying Shared Secret Data.

The initial loading of CAVE for calculation of authentication signatures is given in Exhibit 2-9.

AAV is as defined in §1.1.

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.

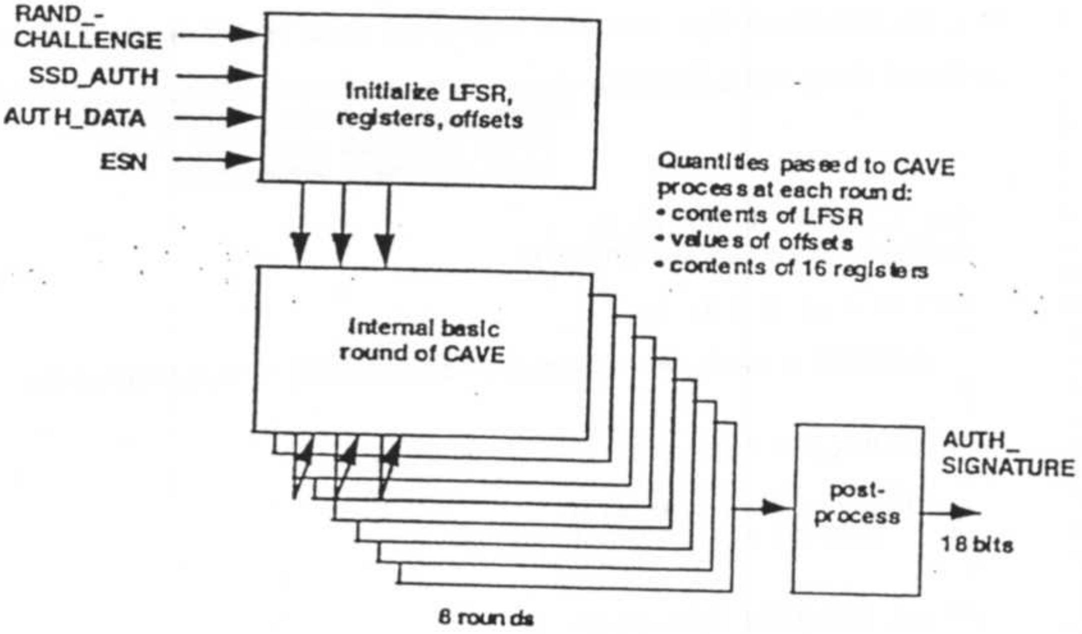
Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

Exhibit 2-11 Code for Calculation of AUTH_SIGNATURE

```
/* Auth_Signature has the same header as CAVE (see Exhibit 2-2) */

unsigned long Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                             const unsigned char AUTH_DATA[3],
                             const unsigned char *SSD_AUTH,
                             const int SAVE_REGISTERS)
{
   int i,offset_1,offset_2;
   unsigned long AUTH_SIGNATURE;

   for (i = 0; i < 4; i++)
   {
      LFSR[i] = RAND_CHALLENGE[i] ^ SSD_AUTH[i] ^ SSD_AUTH[i+4];
   }

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = RAND_CHALLENGE[i];
   }

   /* put SSD_AUTH into r0-r7 */

   for (i = 0; i < 8; i++)
      Register[i] = SSD_AUTH[i];

   Register[8] = AAV;

   /* put AUTH_DATA into r9-r11 */

   for (i = 9; i < 12; i++)
      Register[i] = AUTH_DATA[i-9];

   /* put ESN into r12-r15 */

   for (i = 12; i < 16; i++)
      Register[i] = ESN[i-12];

   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);

   AUTH_SIGNATURE =
      ( ((unsigned long)(Register[0] ^ Register[13]) << 16) +
        ((unsigned long)(Register[1] ^ Register[14]) << 8)  +
        ((unsigned long)(Register[2] ^ Register[15]) )        )
     & 0x3ffff;
```

1

Exhibit 2-10 Calculation of AUTH_SIGNATURE


RAND_-
CHALLENGE
SSD_AUTH
AUTH_DATA
ESN
Initialize LFSR, registers, offsets
Quantities passed to CAVE process at each round:
• contents of LFSR
• values of offsets
• contents of 16 registers
Internal basic round of CAVE
post-process
AUTH_
SIGNATURE
18 bits
8 rounds

2

Exhibit 2-11 Code for Calculation of AUTH_SIGNATURE

```
/* Auth_Signature has the same header as CAVE (see Exhibit 2-2) */

unsigned long Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                             const unsigned char AUTH_DATA[3],
                             const unsigned char *SSD_AUTH,
                             const int SAVE_REGISTERS)
{
   int i,offset_1,offset_2;
   unsigned long AUTH_SIGNATURE;

   for (i = 0; i < 4; i++)
   {
      LFSR[i] = RAND_CHALLENGE[i] ^ SSD_AUTH[i] ^ SSD_AUTH[i+4];
   }

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = RAND_CHALLENGE[i];
   }

   /* put SSD_AUTH into r0-r7 */

   for (i = 0; i < 8; i++)
      Register[i] = SSD_AUTH[i];

   Register[8] = AAV;

   /* put AUTH_DATA into r9-r11 */

   for (i = 9; i < 12; i++)
      Register[i] = AUTH_DATA[i-9];

   /* put ESN into r12-r15 */

   for (i = 12; i < 16; i++)
      Register[i] = ESN[i-12];

   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);

   AUTH_SIGNATURE =
      ( ((unsigned long)(Register[0] ^ Register[13]) << 16) +
        ((unsigned long)(Register[1] ^ Register[14]) << 8)  +
        ((unsigned long)(Register[2] ^ Register[15]) )        )
      & 0x3ffff;
   if (SAVE_REGISTERS)
   {
      /* save LFSR and offsets */

      SAVED_OFFSET_1 = offset_1;
      SAVED_OFFSET_2 = offset_2;
      for (i = 0; i < 4; i++)
      {
         SAVED_LFSR[i] = LFSR[i];
         SAVED_RAND[i] = RAND_CHALLENGE[i];
         SAVED_ESN[i] = ESN[i];
         if (i < 3)
         {
            SAVED_DATA[i] = AUTH_DATA[i];
         }
      }
   }

   return(AUTH_SIGNATURE);
}
```

2.4. 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	32 bits
SAVED_DATA	24 bits
SAVED_ESN	32 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 key for message encryption and the voice privacy mask. Prior to invoking this procedure, the authentication signature calculation procedure (§2.3) must have been invoked with SAVE_REGISTERS set to TRUE. This procedure must be invoked prior to execution of the encryption procedure (§2.5).

The process for generation of CMEA key and voice privacy mask (VPM) will generally be most efficient when concatenated together as described in the following sections (§2.4.1 and §2.4.2). 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 2-10 and Exhibit 2-11 for graphical detail of the generation process.

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

2.4.1. CMEA key Generation

Refer to Exhibit 2-13 or Exhibit 2-14, "CMEA Key Generation and Voice Privacy Mask Generation." Eight bytes of CMEA session key are derived by running CAVE through an 8-round iteration and then two 4-round iterations following an authentication. This is shown in the upper portion of Exhibit 2-13 and Exhibit 2-14. The post-authentication initialization and output processing requirements are as follows:

- First, the LFSR will be re-initialized to the exclusive-or sum of SAVED_LFSR and both halves of SSD_B. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be loaded with SAVED_RAND.
- Second, registers R00 through R07 will be initialized with SSD_B instead of SSD_A.
- Third, Registers R09, R10, and R11 will be loaded with SAVED_DATA.
- Fourth, Registers R12 through R15 will be loaded with SAVED_ESN.
- Fifth, the offset table pointers will begin this process at their final authentication value (SAVED_OFFSET_1 and SAVED_OFFSET_2), rather than being reset to a predetermined state.
- Sixth, the LFSR is loaded before the second and third post-authentication iterations with a "roll-over RAND" comprised of the contents of R00, R01, R14, and R15. If the resulting bit pattern fills the LFSR with all zeroes, then the LFSR will be loaded with SAVED_RAND.

The CMEA key bytes drawn from iterations two and three are labelled:

- k0 = register[4] XOR register[8]; (iteration 2)
- k1 = register[5] XOR register[9]; (iteration 2)
- k2 = register[6] XOR register[10]; (iteration 2)
- k3 = register[7] XOR register[11]; (iteration 2)
- k4 = register[4] XOR register[8]; (iteration 3)
- k5 = register[5] XOR register[9]; (iteration 3)
- k6 = register[6] XOR register[10]; (iteration 3)
- k7 = register[7] XOR register[11]; (iteration 3)

2.4.2. 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 bytes. Each iteration consists of four rounds. The CAVE registers R00 through R15 are not reset between iterations, but

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic In Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data the LFSR is reloaded between iterations with the "rollover RAND" as described in §2.4.1.

The VPM is not to be changed during a call.

Exhibit 2-12 CMEA Key and VPM Generation

```
/* Key_VPM_Generation has the same header as CAVE (see Exhibit 2-2) */

static void roll_LFSR(void)
{
   int i;

   LFSR_A = Register[0];
   LFSR_B = Register[1];
   LFSR_C = Register[14];
   LFSR_D = Register[15];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = SAVED_RAND[i];
   }
}

void Key_VPM_Generation(void)
{
   int i,j,r_ptr,offset_1,offset_2,vpm_ptr;

   /* iteration 1, first pass through CAVE */

   for (i = 0; i < 4; i++)
      LFSR[i] = SAVED_LFSR[i] ^ SSD_B[i] ^ SSD_B[i+4];

   if ((LFSR_A | LFSR_B | LFSR_C | LFSR_D) == 0)
   {
      for (i = 0; i < 4; i++)
         LFSR[i] = SAVED_RAND[i];
   }

   for (i = 0; i < 8; i++)
      Register[i] = SSD_B[i];

   Register[8] = AAV;

   /* put SAVED_DATA into r9-r11 */

   for (i = 9; i < 12; i++)
      Register[i] = SAVED_DATA[i-9];

   /* put ESN into r12-r15 */

   for (i = 12; i < 16; i++)
      Register[i] = ESN[i-12];
```

```
    offset_1 = SAVED_OFFSET_1;
    offset_2 = SAVED_OFFSET_2;

    CAVE(8, &offset_1, &offset_2);

    /* iteration 2, generation of first CMEA key parameters */

    roll_LFSR();
    CAVE(4, &offset_1, &offset_2);
    for (i = 0; i < 4; i++)
       cmeakey[i] = Register[i+4] ^ Register[i+8];

    /* iteration 3, generation of second CMEA key parameters */

    roll_LFSR();
    CAVE(4, &offset_1, &offset_2);
    for (i = 4; i < 8; i++)
       cmeakey[i] = Register[i] ^ Register[i+4];

    /* iterations 4-13, generation of VPM */

    vpm_ptr = 0;
    for (i = 0; i < 10; i++)
    {
       roll_LFSR();
       CAVE(4, &offset_1, &offset_2);
       for (r_ptr = 0; r_ptr < 6; r_ptr++)
       {
          VPM[vpm_ptr] = Register[r_ptr+2] ^ Register[r_ptr+8];
          vpm_ptr++;
       }
    }

    /* iteration 14, generation of last VPM bits */

    roll_LFSR();
    CAVE(4, &offset_1, &offset_2);
    for (j = 0; j < 5; j++)
    {
       VPM[vpm_ptr] = Register[j+2] ^ Register[j+8];
       vpm_ptr++;
    }
}
```

Exhibit 2-13 Generation of CMEA Key and VPM

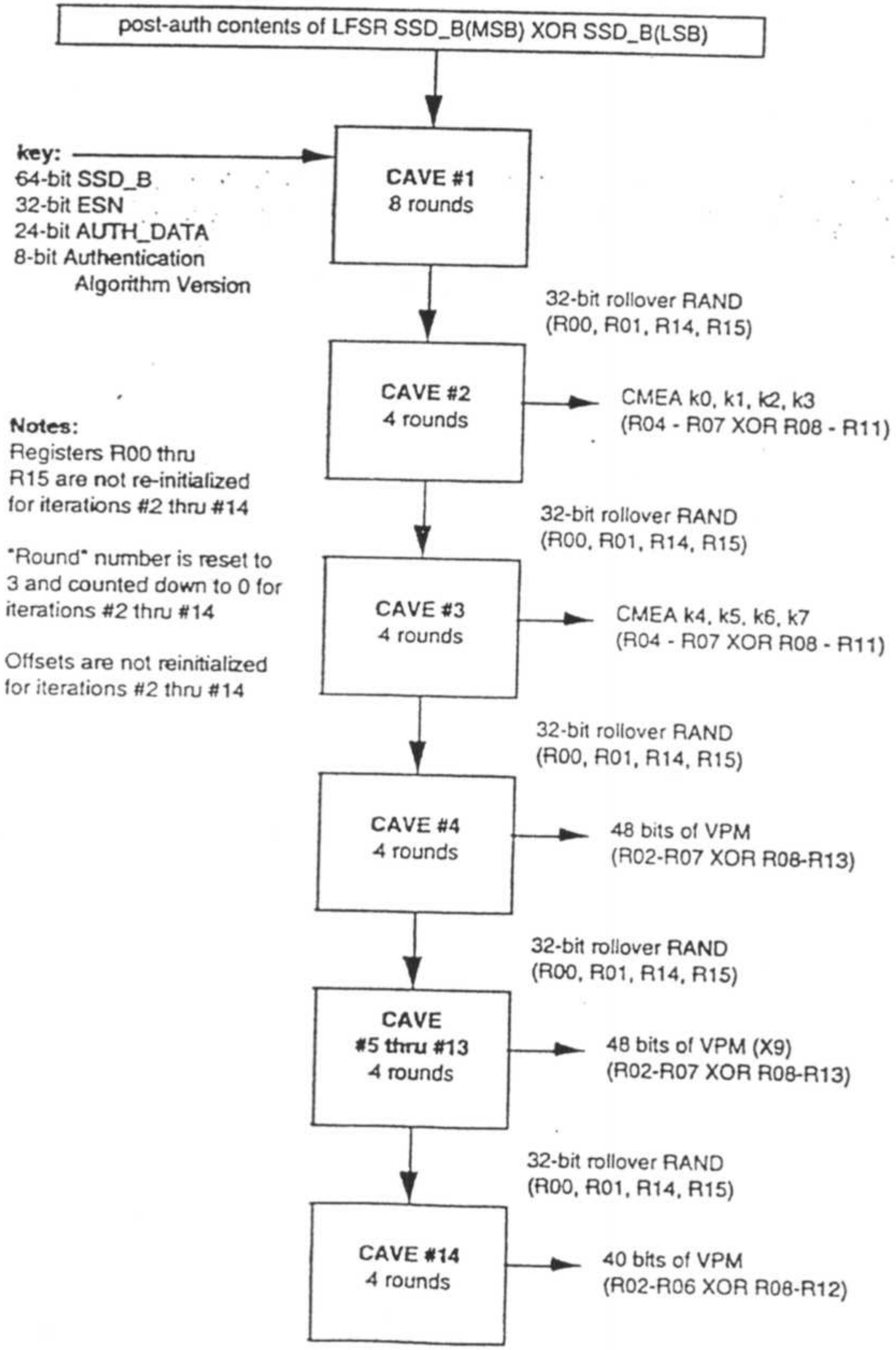
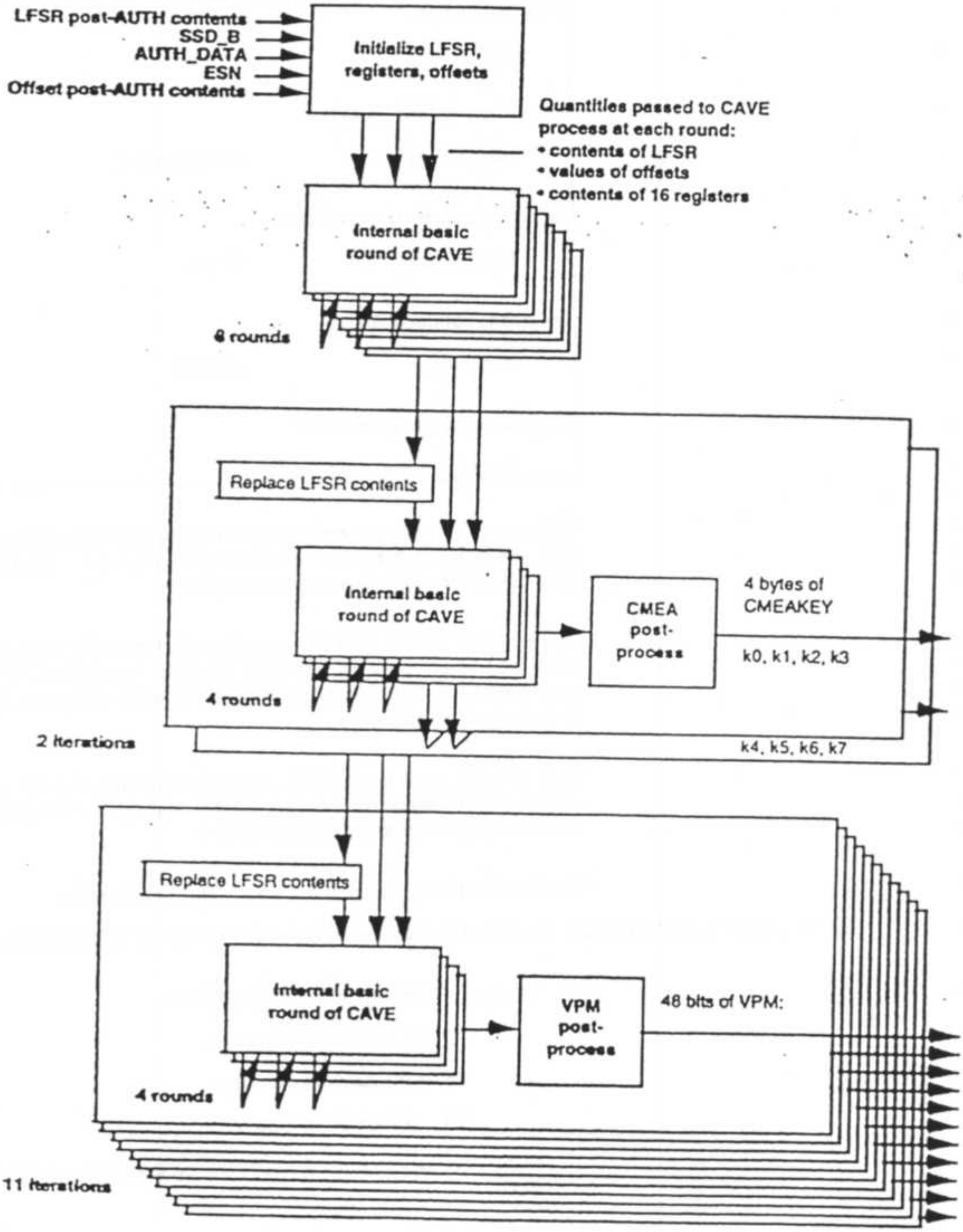

post-auth contents of LFSR SSD_B(MSB) XOR SSD_B(LSB)
key:
64-bit SSD_B
32-bit ESN
24-bit AUTH_DATA
8-bit Authentication Algorithm Version
CAVE #1
8 rounds
32-bit rollover RAND (R00, R01, R14, R15)
CAVE #2
4 rounds
CMEA k0, k1, k2, k3 (R04 - R07 XOR R08 - R11)
Notes:
Registers R00 thru R15 are not re-initialized for iterations #2 thru #14
"Round" number is reset to 3 and counted down to 0 for iterations #2 thru #14
Offsets are not reinitialized for iterations #2 thru #14
32-bit rollover RAND (R00, R01, R14, R15)
CAVE #3
4 rounds
CMEA k4, k5, k6, k7 (R04 - R07 XOR R08 - R11)
32-bit rollover RAND (R00, R01, R14, R15)
CAVE #4
4 rounds
48 bits of VPM (R02-R07 XOR R08-R13)
32-bit rollover RAND (R00, R01, R14, R15)
CAVE #5 thru #13
4 rounds
48 bits of VPM (X9) (R02-R07 XOR R08-R13)
32-bit rollover RAND (R00, R01, R14, R15)
CAVE #14
4 rounds
40 bits of VPM (R02-R06 XOR R08-R12)

Exhibit 2-14 Detailed Generation of CMEA Key and VPM



2.5. CMEA Encryption/Decryption Procedure

Procedure name:	
Encrypt	
Inputs from calling process:	
msg_buf[n]	n*8 bits, n > 1
Inputs from internal stored data:	
CMEAKEY[0-7]	64 bits
Outputs to calling process:	
msg_buf[n]	n*8 bits
Outputs to internal stored data:	
None.	

This algorithm encrypts and decrypts messages that are of length n*8 bits, where n is the number of message bytes, n > 1. Decryption is performed in the same manner as encryption.

The message is first stored in an n-byte buffer called msg_buf[], such that each byte is assigned to one "msg_buf[]" value. msg_buf[] will be encrypted by means of three operations before it is ready for transmission.

This process uses the CMEA eight-byte session key to produce enciphered messages via a unique CMEA algorithm. The process of CMEA key generation is described in §2.4.

The function tbox( ) is frequently used. This is defined as:

tbox(z) = C(((C(((C(((C((z XOR k0)+k1)+z)XOR k2)+k3)+z)XOR k4)+k5)+z)XOR k6)+k7)+z

where "+" denotes modulo 256 addition,

"XOR" is the XOR function,

"z" is the function argument,

k0,. .,k7 are defined above,

and C( ) is the outcome of a CAVE 8-bit table look-up. (Exhibit 2-3)

Exhibit 2-15 shows ANSI C code for an algorithmic procedure for tbox().

Exhibit 2-15 tbox

```
/* tbox has the same header as CAVE (see Exhibit 2-2) */

static unsigned char tbox(const unsigned char z)
{
   int k_index,i;
   unsigned char result;

   k_index = 0;
   result = z;

   for (i = 0; i < 4; i++)
   {
      result ^= cmeakey[k_index];
      result += cmeakey[k_index+1];
      result = z + CaveTable[result];
      k_index += 2;
   }

   return(result);
}
```

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 2-16.

Exhibit 2-16 CMEA Algorithm

```
/* CMEA has the same header as CAVE (see Exhibit 2-2) */

void CMEA(unsigned char *msg_buf, const int byte_count)
{
   int msg_index,half;
   unsigned char k,z;

   /* first manipulation (inverse of third) */

   z = 0;
   for (msg_index = 0; msg_index < byte_count; msg_index++)
   {
      k = tbox(z ^ msg_index);
      msg_buf[msg_index] += k;
      z += msg_buf[msg_index];
   }

   /* second manipulation (self-inverse) */

   half = byte_count/2;
   for (msg_index = 0; msg_index < half; msg_index++)
   {
      msg_buf[msg_index] ^= msg_buf[byte_count - 1 - msg_index] | 0x01;
   }

   /* third manipulation (inverse of first) */
```

```
    z = 0;
    for (msg_index = 0; msg_index < byte_count; msg_index++)
    {
       k = tbox(z ^ msg_index);
       z += msg_buf[msg_index];
       msg_buf[msg_index] -= k;
    }
}
```

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

2.6. Wireless Residential Extension Procedures

This section describes detailed cryptographic procedures for cellular 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.

2.6.1. WIKEY Generation

Procedure name:

WIKEY_Generation

Inputs from calling process:

MANUFACT_KEY	122 bits
PBID	30 bits

Inputs from internal stored data:

AAV	8 bits

Outputs to calling process:

None.

Outputs to internal stored data:

WIKEY	64 bits

This procedure is used to calculate the WIKEY value generated during the manufacturing process. This WIKEY value is stored in semi-permanent memory of the PB.

The initial loading of CAVE for calculation of WIKEY is given in Exhibit 2-17.

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. PB manufactures must provide this number to each ACRE manufacture so that the ACREs can calculate the correct WIKEY values. The 32 MSBs of MANUFACT_KEY must not be all zeroes. There must be at least 40 zeroes and 40 ones in MANUFACT_KEY.

Exhibit 2-17 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 2-18 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-19.

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 2-18 Generation of WIKEY

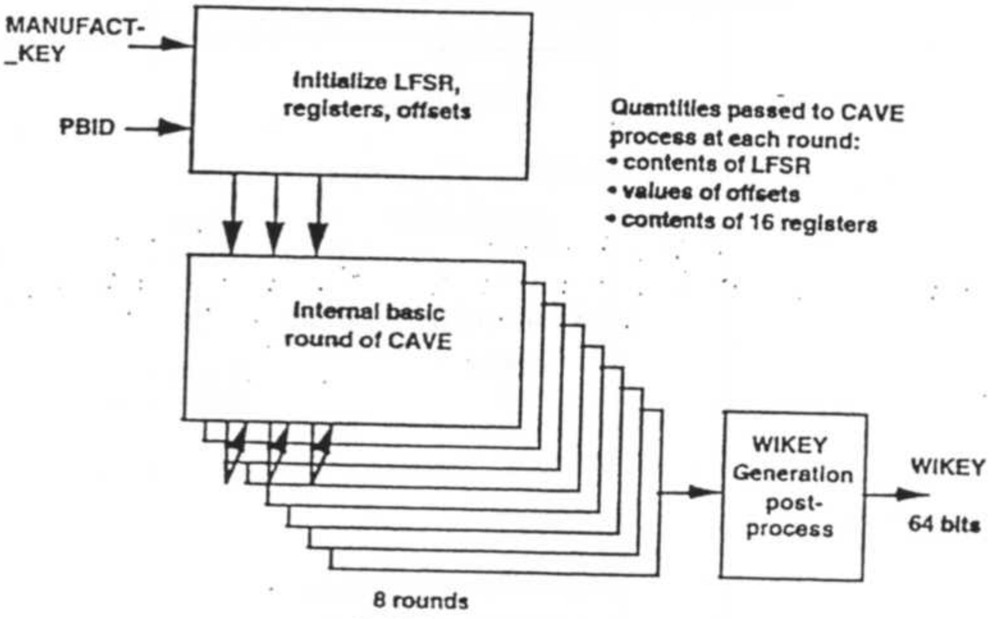

MANUFACT-
_KEY
PBID
Initialize LFSR, registers, offsets
Quantities passed to CAVE process at each round:
• contents of LFSR
• values of offsets
• contents of 16 registers
Internal basic round of CAVE
WIKEY Generation post-process
WIKEY
64 bits
8 rounds

Exhibit 2-19 Code for WIKEY Generation

```
/* WIKEY_Generation has the same header as CAVE (see Exhibit 2-2) */

/* 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
   must be set to 0 by the calling routine. */

void WIKEY_Generation(const unsigned char MANUFACT_KEY[16],
                      const unsigned char PBID[4])
{
   int i,offset_1,offset_2;

   for (i = 0; i < 4; i++)
      LFSR[i] = MANUFACT_KEY[i];
   for (i = 0; i < 8; i++)
      Register[i] = MANUFACT_KEY[i+4];
   Register[8] = AAV;
   for (i = 0; i < 4; i++)
      Register[i+9] = MANUFACT_KEY[i+12];
   Register[12] = Register[12] | PBID[0];
   for (i = 0; i < 3; i++)
      Register[i+13] = PBID[i+1];
   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
   for (i = 0; i < 8; i++)
      WIKEY[i] = Register[i] ^ Register[i+8];
}
```

2.6.2. WIKEY Update Procedure

Procedure name:

WIKEY_Update

Inputs from calling process:

RANDWIKEY 56 bits
PBID 30 bits

Inputs from internal stored data:

WIKEY 64 bits
AAV 8 bits

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 2-20.

Exhibit 2-20 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 2-21 shows the process in graphical form, while the ANSI C for the process is shown in

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

Exhibit 2-22.

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 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 2-21 Generation of WIKEY_NEW

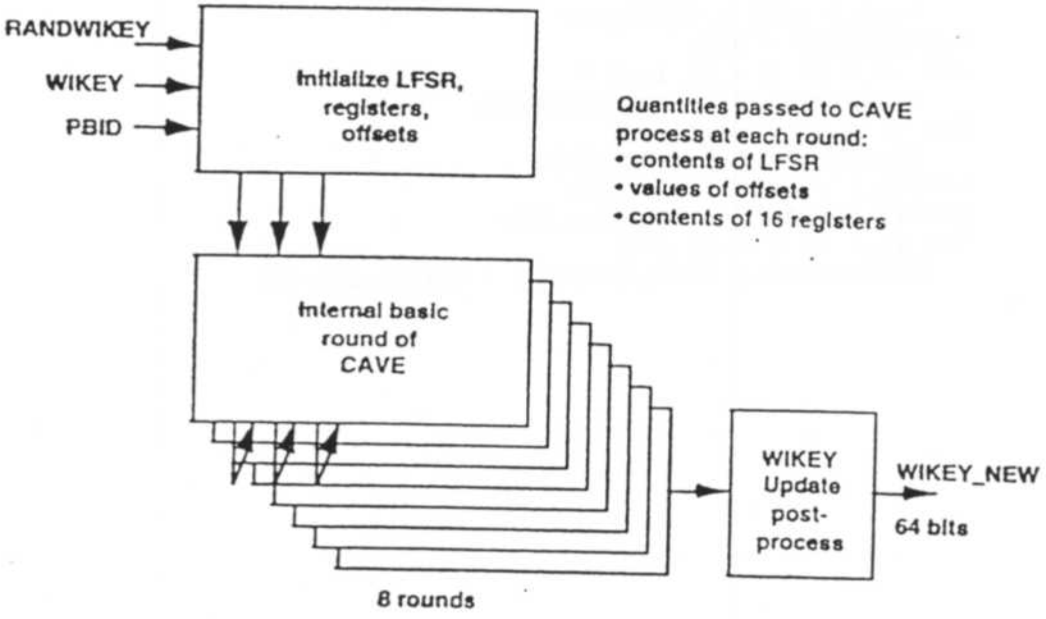

RANDWIKEY
WIKEY
PBID
Initialize LFSR, registers, offsets
Quantities passed to CAVE process at each round:
• contents of LFSR
• values of offsets
• contents of 16 registers
Internal basic round of CAVE
WIKEY Update post-process
WIKEY_NEW
64 bits
8 rounds

Exhibit 2-22 Code for WIKEY_NEW Generation

```
/* WIKEY_Update has the same header as CAVE (see Exhibit 2-2) */

/* Note that PBID is right justified.  This means that the 2 MSBs of PBID
   must be set to 0 by the calling routine. */

void WIKEY_Update(const unsigned char RANDWIKEY[7],
                  const unsigned char PBID[4])
{
   int i,offset_1,offset_2;

   for (i = 0; i < 4; i++)
      LFSR[i] = RANDWIKEY[i+3] ^ WIKEY[i] ^ WIKEY[i+4];
   if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
      for (i = 0; i < 4; i++)
         LFSR[i] = RANDWIKEY[i+3];
   for (i = 0; i < 8; i++)
      Register[i] = WIKEY[i];
   Register[8] = AAV;
   for (i = 0; i < 3; i++)
      Register[i+9] = RANDWIKEY[i];
   for (i = 0; i < 4; i++)
      Register[i+12] = PBID[i];
   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
   for (i = 0; i < 8; i++)
      WIKEY_NEW[i] = Register[i] ^ Register[i+8];
}
```

2.6.3. Wireline Interface Authentication Signature Calculation Procedure

Procedure name:

WI_Auth_Signature

Inputs from calling process:

RAND_CHALLENGE	32 bits
PBID	30 bits
ACRE_PHONE_NUMBER	24 bits

Inputs from internal stored data:

WIKEY	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 WIKEY values.

The initial loading of CAVE for calculation of wireline interface authentication signatures is given in Exhibit 2-23.

For authentication of an ACRE, RAND_CHALLENGE is received from the PB as RAND_ACRE.

For authentication of a PB, RAND_CHALLENGE is received from the ACRE as RAND_PB.

The ACRE_PHONE_NUMBER is 24 bits comprised of the least significant 24 bits of the ACRE's directory number (4 bits per digit). The digits 1 through 9 are represented by their 4-bit binary value (0001b - 1001b), while the digit 0 is represented by 1010b. If the phone number of the acre is less than 6 digits, then the digits are filled on the left with zeros until 6 full digits are reached. Example: If the acre's phone number is (987) 654-3210, ACRE_PHONE_NUMBER is 010101000011001000011010b. If the acre's phone number is 8695, ACRE_PHONE_NUMBER is 000000001000011010010101b.

Exhibit 2-23 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 2-24 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-25.

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 2-24 Calculation of AUTH_SIGNATURE

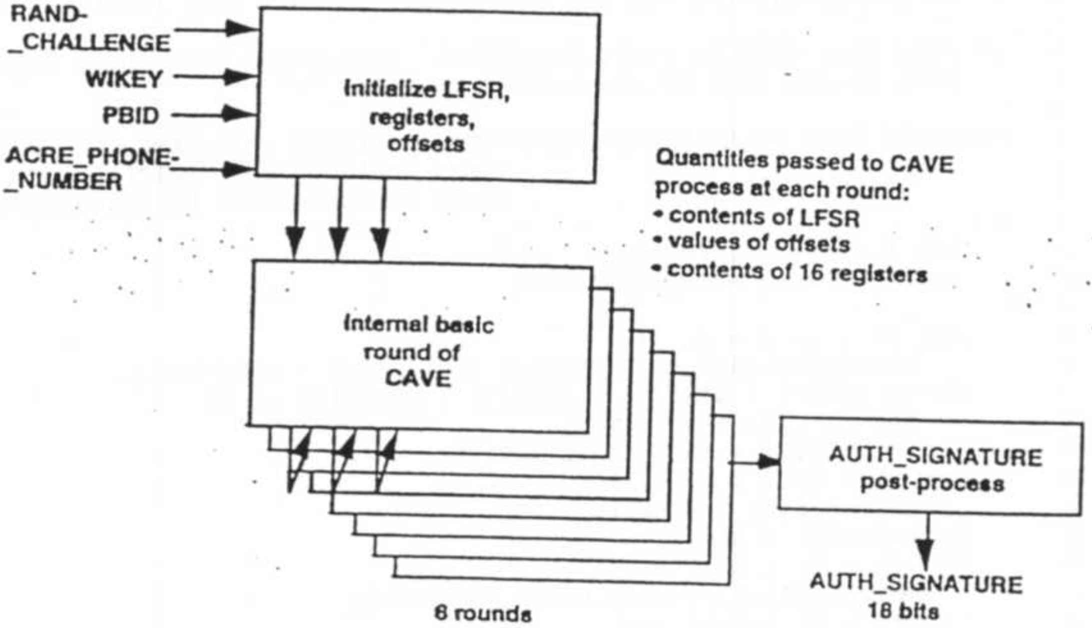

RAND-
_CHALLENGE
WIKEY
PBID
ACRE_PHONE-
_NUMBER
Initialize LFSR, registers, offsets
Quantities passed to CAVE process at each round:
• contents of LFSR
• values of offsets
• contents of 16 registers
Internal basic round of CAVE
8 rounds
AUTH_SIGNATURE post-process
AUTH_SIGNATURE
18 bits

Exhibit 2-25 Code for calculation of AUTH_SIGNATURE

```
/* WI_Auth_Signature has the same header as CAVE (see Exhibit 2-2) */

/* Note that PBID is right justified.  This means that the 2 MSBs of PBID
   must be set to 0 by the calling routine. */

unsigned long WI_Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                                const unsigned char PBID[4],
                                const unsigned char ACRE_PHONE_NUMBER[3])
{
   int i,offset_1,offset_2;
   unsigned long AUTH_SIGNATURE;

   for (i = 0; i < 4; i++)
      LFSR[i] = RAND_CHALLENGE[i] ^ WIKEY[i] ^ WIKEY[i+4];
   if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
      for (i = 0; i < 4; i++)
         LFSR[i] = RAND_CHALLENGE[i];
   for (i = 0; i < 8; i++)
      Register[i] = WIKEY[i];
   Register[8] = AAV;
   for (i = 0; i < 3; i++)
      Register[i+9] = ACRE_PHONE_NUMBER[i];
   for (i = 0; i < 4; i++)
      Register[i+12] = PBID[i];
   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
   AUTH_SIGNATURE =
      (  ((unsigned long)(Register[0] ^ Register[13]) << 16) +
         ((unsigned long)(Register[1] ^ Register[14]) << 8)  +
         ((unsigned long)(Register[2] ^ Register[15]) )           )
      & 0x3ffff;
   return(AUTH_SIGNATURE);
}
```

2.6.4. 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 2-26.

Exhibit 2-26 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

CAVE is run for eight rounds. The 18-bit result is AUTH_SIGNATURE. Exhibit 2-27 shows the process in graphical form, while the ANSI C for the process is shown in Exhibit 2-28.

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 2-27 Calculation of AUTH_SIGNATURE

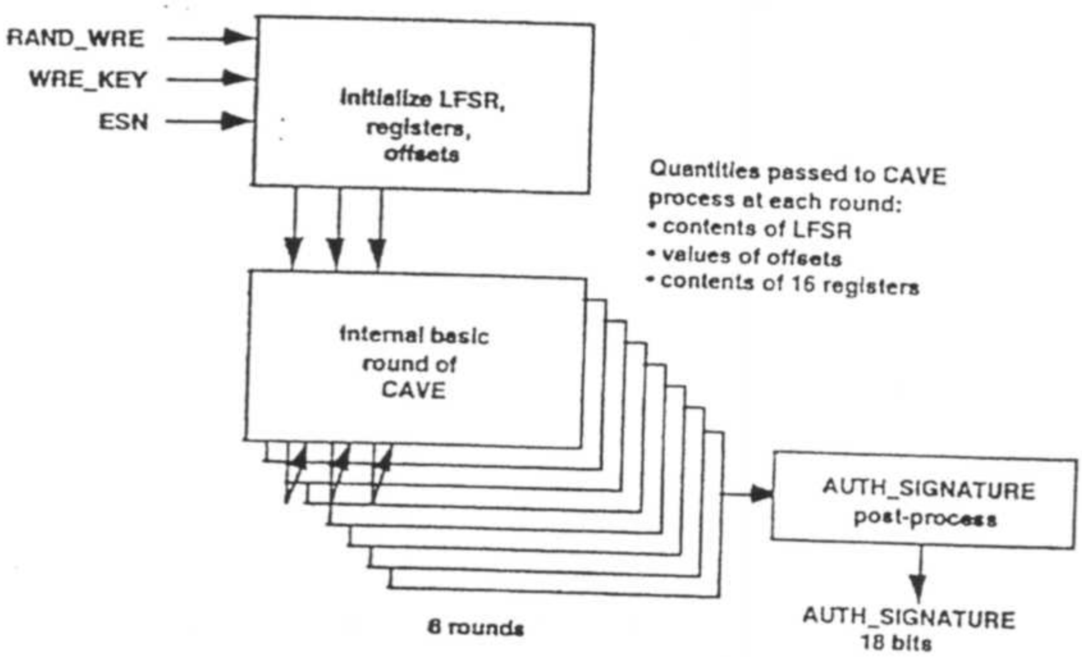

RAND_WRE
WRE_KEY
ESN
Initialize LFSR, registers, offsets
Quantities passed to CAVE process at each round:
• contents of LFSR
• values of offsets
• contents of 16 registers
internal basic round of CAVE
AUTH_SIGNATURE post-process
AUTH_SIGNATURE 18 bits
8 rounds

Exhibit 2-28 Code for calculation of AUTH_SIGNATURE

```
/* WRE_Auth_Signature has the same header as CAVE (see Exhibit 2-2) */

/* Note that RAND_WRE is left justified and PBID is right justified.
   This means that the 5 LSBs of RAND_WRE and the 2 MSBs of PBID
   must be set to 0 by the calling routine. */

unsigned long WRE_Auth_Signature(const unsigned char RAND_WRE[3],
                                 const unsigned char PBID[4],
                                 const unsigned char ESN[4])
{
   int i,offset_1,offset_2;
   unsigned long AUTH_SIGNATURE;

   for (i = 0; i < 3; i++)
      LFSR[i] = RAND_WRE[i];
   LFSR[2] = LFSR[2] | (PBID[2] & 0x1F);
   LFSR[3] = PBID[3];
   for (i = 0; i < 4; i++)
      LFSR[i] = LFSR[i] ^ WRE_KEY[i] ^ WRE_KEY[i+4];
   if ((LFSR[0] | LFSR[1] | LFSR[2] | LFSR[3]) == 0)
   {
     for (i = 0; i < 3; i++)
        LFSR[i] = RAND_WRE[i];
     LFSR[2] = LFSR[2] | (PBID[2] & 0x1F);
     LFSR[3] = PBID[3];
   }
   for (i = 0; i < 8; i++)
      Register[i] = WRE_KEY[i];
   Register[8] = AAV;
   for (i = 0; i < 3; i++)
      Register[i+9] = PBID[i];
   for (i = 0; i < 4; i++)
      Register[i+12] = ESN[i];
   offset_1 = offset_2 = 128;
   CAVE(8, &offset_1, &offset_2);
   AUTH_SIGNATURE =
      (  ((unsigned long)(Register[0] ^ Register[13]) << 16) +
         ((unsigned long)(Register[1] ^ Register[14]) << 8)  +
         ((unsigned long)(Register[2] ^ Register[15]) )          )
      & 0x3ffff;
   return(AUTH_SIGNATURE);
}
```

2.7. Cellular Data Encryption

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

In this description, a "byte" means "octet" and has 8 bits. The left-most bit or byte of a numerical quantity is the numerically most significant, the right-most bit or byte is the numerically least significant. Bit positions in a multibit field (byte or 32 bit quantity) are numbered from 1 for the left-most up through 8 or 32 for the right-most bit. If x is a multi-byte quantity, high(x) denotes the left-most byte of x.

The ORYX key byte generator maintains three 32 bit registers, K, A, and B and a 256-byte look up table L, whose i-th entry is denoted L[i], for $0 \leq i < 256$.

Register K is a 32-bit linear feedback "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 thereby clearing the left-most bit) and XORing the bit shifted out of the right-most position into bit positions 28, 19, 18, 16, 14, 11, 10, 9, 6, 5, and 0, when counting the bit positions with 0 as the left-most bit and 31 as the right-most bit.

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.

The polynomial k(z) is primitive and has Peterson & Weldon octal code 42003247143.

Registers A and B are 32 bit linear feedback Galois shift registers, shifting to the left: the leftmost bit is XORed into various bit positions. Register A sometimes steps according to recursion polynomial

$$a_1(z) = z^{32} + z^{26} + z^{23} + z^{22} + z^{16} + z^{12} + z^{11} + z^{10} + z^{8} + z^{7} + z^{5} + z^{4} + z^{2} + z + 1$$

and sometimes according to recursion polynomial

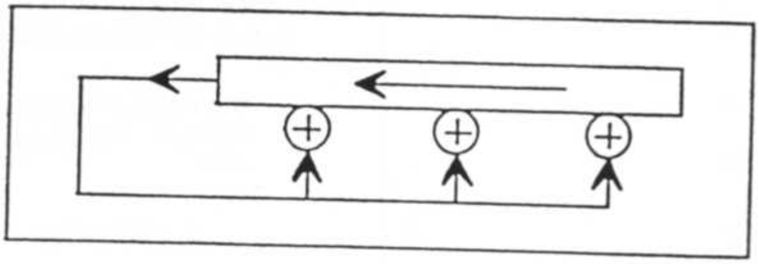
$$a_2(z) = z^{32} + z^{27} + z^{26} + z^{25} + z^{24} + z^{23} + z^{22} + z^{17} + z^{13} + z^{11} + z^{10} + z^{9} + z^{8} + z^{7} + z^{2} + z + 1$$

The decision is based on the current high order bit of K. First K is stepped. If the (new) high order bit of K is set, register A steps according to polynomial $a_1(z)$; if the high order bit of K is clear, register A steps according to polynomial $a_2(z)$. The $a_1(z)$ shift is implemented as follows: the 32 bits of A are shifted to the left, clearing the right-most bit. The bit shifted out of the left-most position of A is XORed into bit positions 26, 23, 22, 16, 12, 11, 10, 8, 7, 5, 4, 2, 1, and 0, when counting the bit positions so 0 is the right-most bit and 31 the left-most. The $a_2(z)$ shift is the same, except the bit shifted out of the left-most position is XORed into positions 27, 26, 25, 24, 23, 22, 17, 13, 11, 10, 9, 8, 7, 2, 1, and 0. Then register B steps once (if the next-to-high order bit of K is clear) or twice (if the next-to-high order bit of K is set) according to recursion polynomial

$$\begin{aligned} b(z) &= (z+1)(z^{31}+z^{20}+z^{15}+z^{5}+z^{4}+z^{3}+1) \\ &= z^{32}+z^{31}+z^{21}+z^{20}+z^{16}+z^{15}+z^{6}+z^{3}+z+1 \end{aligned}$$

This is also implemented with a left shift, XORing the shifted out bit into bit positions 31, 21, 20, 16, 15, 6, 3, 1, and 0. Polynomials $a_1(z)$, $a_2(z)$, and the degree 31 factor of polynomial $b(z)$ are all primitive, with Peterson & Weldon octal codes 40460216667, 41760427607, and 20004100071, respectively.

Exhibit 2-29 A Galois Stepping Shift Register



Here is how DataKey is formed from SSD-B, using ORYX as a hash function: First, register A is initialized with the first 32 bits of SSD-B, B is initialized with the remaining 32 bits of SSD-B and K is initialized with the XOR of A and B. Then K is stepped 256 times.

After the i-th step, for $0 \leq i < 256$, the i-th entry, L[i], in the look up table is initialized with the current value of K_0. Then the following three-step procedure is repeated 32 times:

1. ORYX is stepped, producing a key byte, which is temporarily stored in byte x. Register A is modified by shifting its contents to the left by 9 bits, so that what was in bit positions 10 through 32 now occupies positions 1 through 23. Bit position 24 is cleared. The value of x is stored in bit positions 25 through 32.

2. ORYX is stepped, producing a key byte, which is temporarily stored in byte x. Register B is modified by shifting its contents to the left by 9 bits, so what was in bit positions 10 through 32 now occupies positions 1 through 23. Bit position 24 is cleared. The value of x is stored in bit positions 25 through 32.

3. ORYX is stepped, producing a key byte, which is temporarily stored in byte x. The value of x is stored in bit positions 9 through

16 of K. The value of x is ANDed into bit positions 17 through 24 of K.

The XOR of the final values of K, A, and B is stored in DataKey.

All data streams in a call share the same look up table L, which is initialized as follows:

K is set equal to RAND_S.

The i-th cell in the L table, L[i] is initialized with the value i, for $0 \leq i < 256$.

Then, the K register is stepped 256 times. After the i-th step, for $0 \leq i < 256$, the values stored in the high(K)-th cell and the i-th cells of the L table are interchanged.

Once the DataKey and the look up table L have been initialized, ORYX is ready to produce key bytes.

The key bytes in a frame mask are produced by initializing the registers K, A, and B with values derived from DataKey and HOOK as follows.

1. K is set equal to the current value of HOOK. If K_1, K_2, K_3, and K_4 denote the four bytes of K, the following assignments are made in turn:

$$K_1 = L[K_1 + K_4]$$

$$K_2 = L[K_2 + K_4]$$

$$K_3 = L[K_3 + K_4]$$

$$K_4 = L[K_4]$$

where the additions $K_i + K_4$ are performed modulo 256. Then K is stepped once. A is set equal to DataKey XOR-ed with K.

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

3. 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 bytes are the n bytes of the frame mask.

2.7.1. Data Encryption Key Generation Procedure

Procedure name:

DataKey_Generation

Inputs from calling process:

RAND	32 bits

Inputs from internal stored data:

SSD_B	64 bits

Outputs to calling process:

None.

Outputs to internal stored data:

DataKey	32 bits
L	256*8 bits

This algorithm generates DataKey, a period key used for generation of encryption masks for cellular data, and L, a table used in mask generation.

The DataKey_Generation procedure is executed at the beginning of each call, using the values of SSD_B and RAND in effect at the start of the call. The values of DataKey and L shall not change during a call.

The calculation of DataKey depends only on SSD_B, and may be computed and saved when SSD is updated. The calculation of L depends on RAND, and shall be performed at the beginning of each call.

Exhibit 2-30 describes the calculation in ANSI C.

Exhibit 2-30 DataKey Generation

```
/* header for DataKey_Generation */

unsigned long K;                    /* 32-bit K  register */
unsigned long A, B;                 /* 32-bit LFSRs */
unsigned char L[256];               /* look up table */
#define high(x) (unsigned char)(0xffU&(x>>24))    /* leftmost byte */
#define FA1 000460216667            /* Peterson & Weldon prim 32 */
#define FA2 001760427607            /* Peterson & Weldon prim 32 */
#define FB 020014300113             /* P&W prim 31 020004100071 times z+1 */
#define FK 030634530010             /* reverse of P&W prim 32 042003247143 */
unsigned char SSD_B[8];
unsigned long DataKey;              /* data encryption key */

void kstep(void);
void DataKey_Generation(const unsigned char [] );
unsigned char keygen(void);
void Data_Mask(const unsigned long , unsigned char [], const int );

/* end of header */

void DataKey_Generation(const unsigned char RAND[4])
{
   int i,j;
   unsigned long temp;
   unsigned char tempc;

   A = 0;
   for(i=0; i<4; i++)
      A = (A<<8) + (unsigned long)SSD_B[i];
   B = 0;
   for(i=4; i<8; i++)
      B = (B<<8) + (unsigned long)SSD_B[i];

   K = A ^ B;
   for(i=0; i<256; i++)
   {
      kstep();
      L[i] = high(K);
   }
   for(i=0; i<32; i++)
   {
      temp = (unsigned long)keygen();
      A = (A<<9) + temp;
      temp = (unsigned long)keygen();
      B = (B<<9) + temp;
      temp = (unsigned long)keygen();
      K = (0xff00ffffU & K) + (temp << 16);
      K &= 0xffff00ffU + (temp<<8);
   }
   DataKey = A ^ B ^ K;
   DataKey &= 0xffffffff;      /* use only 32 bits */
```

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

```
    /* make L table */
    K = 0;
    for(i=0; i<4; i++)
       K = (K<<8) + (unsigned long)RAND[i];
    for (i=0; i<256; i++)
       L[i] = (unsigned  char)i;

    /* use high byte of K to permute 0 through 255 */
    for (i=0; i< 256; i++)
    {
       kstep();
       j = high(K);
       tempc = L[i];
       L[i] = L[j];
       L[j] = tempc;
    }
}
unsigned char keygen(void)
{
   unsigned char x;
   int  i, trips;

   kstep();

   /*
    * if high bit of K set, use A1 feedback
    * otherwise use A2 feedback
    */
   if((1UL<<31) & A)
   {
      A += A;
      if((1UL<<31) & K)
         A = A ^ FA1;
      else
         A = A ^ FA2;
   }
   else
      A += A;

   /*
    * if next-high bit of  K set, step B twice
    * otherwise once
    */
   if((1UL<<30) & K)
      trips = 2;
   else
      trips = 1;
   for(i=0; i<trips; i++)
   {
      if((1UL<<31) & B)
      {
         B += B;
         B = B ^ FB;
      }
      else
         B += B;
   }
```

```
    x = high(K) + L[high(A)] + L[high(B)];
    x &= 0xffU;
    return x;                                    /* use only 8 bits */
}
/*
 * step the K register
 */
void kstep(void)
{
    if(K==0) K = 0x31415926;
    if(K&1){
        K = (K>>1) ^ FK;
    } else
        K = (K>>1);
    K &= 0xffffffff;
}
```

2.7.2. Data Encryption Mask Generation Procedure

Procedure name:	
Data_Mask	
Inputs from calling process:	
HOOK	32 bits
mask	array pointer
len	integer
Inputs from internal stored data:	
SSD_B	64 bits
Outputs to calling process:	
mask	len*8 bits
Outputs to internal stored data:	
None.	

This algorithm generates an encryption mask of length len*8 bits, where n is the number of mask bytes.

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

- The bits of the HOOK variable that change most frequently shall be placed in the least significant octet.
- No value of HOOK shall be used more than once during a call.
- Mask bytes produced using a value of HOOK shall be used to encrypt only one set of data bytes.
- Mask bytes generated using a value of HOOK shall not be used for encryption of data sent in different directions of transmission nor for data sent on different logical channels in any direction of transmission.

Exhibit 2-31 describes the calculation in ANSI C.

Exhibit 2-31 Data Encryption Mask Generation

```
/* Data_Mask has the same header as DataKey_Generation
   (see Exhibit 2-30) */

void Data_Mask(const unsigned long HOOK,
               unsigned char mask[],
               const int len)
{
  int i;

  K = (unsigned long)L[HOOK&0xff];
  K += ((unsigned long)L[((HOOK>>8)+HOOK)&0xff])<<8;
  K += ((unsigned long)L[((HOOK>>16)+HOOK)&0xff])<<16;
  K += ((unsigned long)L[((HOOK>>24)+HOOK)&0xff])<<24;
  kstep(); A = DataKey ^ K; /* kstep() is defined in Exhibit 2-30 */
  kstep(); B = DataKey ^ K;
  kstep(); K = DataKey ^ K;

  for(i=0; i<len; i++)
    mask[i] = keygen(); /* keygen() is defined in Exhibit 2-30 */
}
```

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			

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
CMEA output         =  E5 6B 5F 01 65 C6

VPM = 18 93 94 82 4A 1A 2F 99
      A5 39 F9 5B 4D 22 D5 7C
      EE 32 AC 21 6B 26 0D 36
      A7 C9 63 88 57 8C B9 57
      E2 D6 CA 1D 77 B6 1F D5
      C7 1A 73 A4 17 B2 12 1E
      95 34 70 E3 9B CA 3F D0
      50 BE 4F D6 47 80 CC B8
      DF
```

3.1.2. Vector 2

If RAND_CHALLENGE = 5375 DF99:

(Using SSD_AUTH = SSD_A)

```
AUTH_SIGNATURE=    0 255A
CMEA key k0,. .k7    =  F0 06 A8 5A 05 CD B3 2A
CMEA output          =  2B AD 16 A9 8F 32


VPM = 20 38 01 6B 89 3C F8 A0
      28 48 98 75 AB 18 65 5A
      49 6E 0B BB D2 CB A8 28
      46 E6 D5 B4 12 B3 8C 9E
      76 6C 9E D4 98 C8 A1 4A
      D2 DC 94 B0 F6 D4 3E E0
      D1 6C 7E 9E AC 6B CA 43
      02 C9 23 63 6F 61 68 E8
      8F
```

3.1.3. Test Program

```
#include <stdio.h>

void CAVE(int number_of_rounds,
          int *offset_1,
          int *offset_2);

void A_Key_Checksum(const char A_KEY_DIGITS[20],
                    char A_KEY_CHECKSUM[6]);

int A_Key_Verify(char A_KEY_DIGITS[26]);

void SSD_Generation(unsigned char RANDSSD[7]);

unsigned long Auth_Signature(unsigned char RAND_CHALLENGE[4],
                             unsigned char AUTH_DATA[3],
                             unsigned char *SSD_AUTH,
                             int SAVE_REGISTERS);

void Key_VPM_Generation(void);

void CMEA(unsigned char *msg_buf, int byte_count);

extern unsigned char       A_key[8];
extern unsigned char    SSD_A[8];
extern unsigned char    SSD_B[8];

extern unsigned char    cmeakey[8];

extern unsigned char    VPM[65];
```

```
/*Test vector inputs and results */

extern unsigned char ESN[4];
unsigned char RANDSSD[7] = { 0x4d, 0x18, 0xee, 0xaa,
                             0x05, 0x89, 0x5c };
unsigned char MIN1[3] =    { 0x79, 0x29, 0x71 };
unsigned char MIN2[2] =    { 0x02, 0x8d };

unsigned char cmeakey1[8] = { 0xa0, 0x7b, 0x1c, 0xd1,
                              0x02, 0x75, 0x69, 0x14 };
unsigned char cmeakey2[8] = { 0xf0, 0x06, 0xa8, 0x5a,
                              0x05, 0xcd, 0xb3, 0x2a };

unsigned char RAND1[4]    = { 0x34, 0xa2, 0xb0, 0x5f };
unsigned char RAND2[4]    = { 0x53, 0x75, 0xdf, 0x99 };

unsigned char buf[6] = { 0xb6, 0x2d, 0xa2, 0x44, 0xfe, 0x9b };

/* entered A_key and checksum */

char digits[26] =
   { '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' };

/* CAVE outputs */

extern unsigned char       A_key[8];
extern unsigned char   SSD_A_NEW[8];
extern unsigned char   SSD_B_NEW[8];
unsigned long          AUTHR;

/* CMEA keys */

extern unsigned char cmeakey[8];
extern unsigned char VPM[65];

void SSD_Update(void)
{
   int i;

   for (i = 0; i < 8; i++)
   {
      SSD_A[i] = SSD_A_NEW[i];
      SSD_B[i] = SSD_B_NEW[i];
   }
}
```

```
void main(void)
{
   int i, j;
   unsigned char auth_data[3],test_buf[6];

   /* assign ESN value */

   ESN[0] = 0xd7;
   ESN[1] = 0x5a;
   ESN[2] = 0x96;
   ESN[3] = 0xec;

   /* check A_key and SSD */

   if(A_Key_Verify(digits))
   {
      printf("A key verified ok\n");
   }
   else
   {
      printf("A key verification failed\n");
      return;
   }

   /* check SSD generation process */

   SSD_Generation(RANDSSD);
   SSD_Update();

   printf("SSD_A =");
   for (i = 0; i < 4; i++)
   {
      printf(" ");
      for (j = 0; j < 2; j++)
      {
         printf("%02x",(unsigned int)SSD_A[2*i+j]);
      }
   }
   printf("\n");

   printf("SSD_B =");
   for (i = 0; i < 4; i++)
   {
      printf(" ");
      for (j = 0; j < 2; j++)
      {
         printf("%02x",(unsigned int)SSD_B[2*i+j]);
      }
   }
   printf("\n");

   /* Inputs for test vectors */

   /* put MIN1 into auth_data (no dialed digits for this test) */

   for (i = 0; i < 3; i++)
      auth_data[i] = MIN1[i];

```

```
    /* vector 1 */

    printf("\nVector 1\n\n");

    AUTHR = Auth_Signature(RAND1,auth_data,SSD_A,1);

    printf("RAND_CHALLENGE     =");
    for (i = 0; i < 2; i++)
    {
       printf(" ");
       for (j = 0; j < 2; j++)
       {
          printf("%02x",(unsigned int)RAND1[2*i+j]);
       }
    }
    printf("\n");

    printf("AUTH_SIGNATURE     = %01lx %04lx\n", AUTHR >> 16, AUTHR &
0x0000ffff);

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];

    Key_VPM_Generation();
    CMEA(test_buf,6);

    printf("CMEAkey k0,.  .,k7 =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)cmeakey[i]);
    printf("\n");

    printf("CMEA Output        =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    printf("VPM:");
    for (i = 0; i < 65; i++)
    {
       printf(" %02x",(unsigned int)VPM[i]);
       if (((i+1)%8) == 0)
          printf("\n    ");
    }
    printf("\n");

    /* vector 2 */

    printf("\nVector 2\n\n");

    AUTHR = Auth_Signature(RAND2,auth_data,SSD_A,1);
```

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

```
    printf("RAND_CHALLENGE     =");
    for (i = 0; i < 2; i++)
    {
       printf(" ");
       for (j = 0; j < 2; j++)
       {
          printf("%02x",(unsigned int)RAND2[2*i+j]);
       }
    }
    printf("\n");

    printf("AUTH_SIGNATURE      = %01lx %04lx\n", AUTHR >> 16, AUTHR &
0x0000ffff);

    for (i = 0; i < 6; i++)
       test_buf[i] = buf[i];

    Key_VPM_Generation();
    CMEA(test_buf,6);

    printf("CMEAkey k0,.  .,k7 =");
    for (i = 0; i < 8; i++)
       printf(" %02x",(unsigned int)cmeakey[i]);
    printf("\n");

    printf("CMEA Output        =");
    for (i = 0; i < 6; i++)
       printf(" %02x",(unsigned int)test_buf[i]);
    printf("\n");

    printf("VPM:");
    for (i = 0; i < 65; i++)
    {
       printf(" %02x",(unsigned int)VPM[i]);
       if (((i+1)%8) == 0)
          printf("\n    ");
    }
    printf("\n");
}
```

3.2. Wireless Residential Extension Test Vectors

3.2.1. Input data

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

3.2.2. Test Program

```
#include <stdio.h>

void WIKEY_Generation(const unsigned char MANUFACT_KEY[16],
                      const unsigned char PBID[4]);

void WIKEY_Update(const unsigned char RANDWIKEY[7],
                  const unsigned char PBID[4]);

unsigned long WI_Auth_Signature(const unsigned char RAND_CHALLENGE[4],
                                const unsigned char PBID[4],
                                const unsigned char ACRE_PHONE_NUMBER[3]);

unsigned long WRE_Auth_Signature(const unsigned char RAND_WRE[3],
                                 const unsigned char PBID[4],
                                 const unsigned char ESN[4]);

/* Test vector inputs */

unsigned char manufact[16] = { 0x85, 0x03, 0xA7, 0xDC, 0x14, 0x35, 0xFA, 0x90,
                               0xB6, 0x72, 0x40, 0x32, 0x45, 0x05, 0x33, 0xC0 };

unsigned char baseid[4] = { 0x00, 0x00, 0x01, 0x00 };

unsigned char random_challenge[4] = { 0x7E, 0x49, 0xAE, 0x4F };

unsigned char acre_phone[3] = { 0x49, 0x85, 0xA6 };

unsigned char random_wre[3] = { 0x62, 0xEA, 0x40 };
unsigned char hs_esn[4] = { 0xED, 0x07, 0x13, 0x95 };

unsigned char rand_wikey[7] = { 0xB7, 0xFC, 0x75, 0x5A, 0xF0, 0xA4, 0x90 };

/* CAVE outputs */

extern unsigned char    WIKEY[8];
extern unsigned char    WIKEY_NEW[8];
extern unsigned char    WRE_KEY[8];

void main(void)
{
    int i;
    unsigned long auth_sig;

    WIKEY_Generation(manufact,baseid);
    printf("WIKEY = ");
    for(i=0;i<8;i++)
      printf("%02x",(unsigned int)WIKEY[i]);
    printf("\n");

    auth_sig = WI_Auth_Signature(random_challenge,baseid,acre_phone);
    printf("AUTH_SIGNATURE = %05lx\n",auth_sig);

    WRE_KEY[0] = 0xCB;
    WRE_KEY[1] = 0x60;
    WRE_KEY[2] = 0xF9;
    WRE_KEY[3] = 0x9F;
    WRE_KEY[4] = 0x5B;
    WRE_KEY[5] = 0x15;
    WRE_KEY[6] = 0x6F;
    WRE_KEY[7] = 0xAE;

    auth_sig = WRE_Auth_Signature(random_wre,baseid,hs_esn);
    printf("AUTH_SIGNATURE = %05lx\n",auth_sig);

    WIKEY_Update(rand_wikey,baseid);
    printf("WIKEY_NEW = ");
    for(i=0;i<8;i++)
      printf("%02x",(unsigned int)WIKEY_NEW[i]);
    printf("\n");
}
```

3.2.3. Test Program Output

```
WIKEY = cb60f99f5b156fae
AUTH_SIGNATURE = 2cf01
AUTH_SIGNATURE = 12893
WIKEY_NEW = 167ca928358cceba
```

Information disclosed in this document is subject to the export jurisdiction of the US Department of State as specified in International Traffic in Arms Regulations (title 22 CFR parts 120 through 130 inclusive). A license issued by the Department of State is required for the export of such technical data.

3.3. Data Encryption Test Vector

3.3.1. Input data

```
SSD_B= 1492 5280 1776 1867
RAND = 1234 ABCD
HOOK = CDEF 5678
```

24 bytes of mask to be returned

3.3.2. Test Program

```
#include <stdio.h>

extern unsigned char L[256];
extern unsigned char SSD_B[8];
extern unsigned long DataKey;

void DataKey_Generation(const unsigned char [] );
void Data_Mask(const unsigned long , unsigned char [], const int );

void main(void)
{
   int i, j;
   unsigned long hook;
   unsigned char buf[24], rand[4];

   rand[0] = 0x12;
   rand[1] = 0x34;
   rand[2] = 0xab;
   rand[3] = 0xcd;

   hook = 0xcdef5678;

   SSD_B[0] = 0x14;
   SSD_B[1] = 0x92;
   SSD_B[2] = 0x52;
   SSD_B[3] = 0x80;
   SSD_B[4] = 0x17;
   SSD_B[5] = 0x76;
   SSD_B[6] = 0x18;
   SSD_B[7] = 0x67;

   printf("\nSSD_B =");
   for (i = 0; i < 4; i++)
   {
      printf(" ");
      for (j = 0; j < 2; j++)
      {
         printf("%02x", (unsigned int)SSD_B[2*i+j]);
      }
   }

```

```
      printf("\nRAND =");
      for (i = 0; i < 2; i++)
      {
         printf(" ");
         for (j = 0; j < 2; j++)
         {
            printf("%02x", (unsigned int)rand[2*i+j]);
         }
      }

      printf("\nHOOK = %04lx %04lx", hook >> 16, hook & 0x0000ffff);

      printf("\n24 bytes of mask to be returned");

      DataKey_Generation(rand);

      printf("\n\nOutput:\n\n");

      printf("\nDataKey = %04lx %04lx\n", DataKey >> 16, DataKey &
 0x0000ffff);

      printf("\n\nL:\n\n");

      for(i = 0; i < 16; i++)
      {
         for (j = 0; j < 16; j++)
         {
            printf("%02x ", (unsigned int)L[16*i+j]);
         }
         printf("\n");
      }

      Data_Mask(hook, buf, 24);

      printf("\n\nmask:\n\n");

      for(i = 0; i < 2; i++)
      {
         for (j = 0; j < 12; j++)
         {
            printf("%02x ", (unsigned int)buf[12*i+j]);
         }
         printf("\n");
      }
}
```

3.3.3. Test Program Output

DataKey = 8469 B522

L:

```
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
17 A8 78 46 90 C0 41 BF 94 97 D3 43 01 C8 AB DD
8A 1C BB 08 F6 4C 4B 27 28 1A 03 C4 FA E7 B5 A2
EB B3 C9 72 52 A0 0A E9 D8 C6 3F AF 05 CA C3 AE
9E 9A EF B7 8D E6 A4 D5 82 F3 77 54 42 B2 18 73
E1 DC BD B9 3E 37 59 CD EC 02 80 81 AC 2A 31 EA
89 1E 63 D6 91 92 D4 11 EE 9C 12 A5 A6 3A C2 35
F5 67 CF 45 44 DB 22 FE 55 C7 56 B1 AD F4 F9 57
F8 DF 1D 58 9B 34 ED 0B D7 AA 99 7A C1 7C E0 E3
5B 5C 21 61 85 19 84 D0 3C 26 87 98 B0 F7 23 24
```

mask

```
57 F6 C2 03 7C 78 2F CC 8B 3E E4 0B
E0 4D 73 80 FF 2A 4D 2F 8D 74 8E DB
```