JunHyuk Song
Radha Poovendran
University of Washington
Jicheol Lee
Samsung Electronics
Tetsu Iwata
Ibaraki University

INTERNET DRAFT
Expires: June 8, 2006
December 9 2005

The AES-CMAC Algorithm
draft-songlee-aes-cmac-03.txt

Status of This Memo

Copyright Notice

Abstract

National Institute of Standards and Technology (NIST) has newly specified the Cipher-based Message Authentication Code (CMAC) which is equivalent to the One-Key CBC MAC1 (OMAC1) submitted by Iwata and Kurosawa. This memo specifies the authentication algorithm based on CMAC with 128-bit Advanced Encryption Standard (AES).
This new authentication algorithm is named AES-CMAC.
The purpose of this document is to make the AES-CMAC algorithm conveniently available to the Internet Community.

Table of Contents

1. Introduction 2
2. Specification of AES-CMAC 3
2.1 Basic definitions 3
2.2 Overview 4
2.3 Subkey Generation Algorithm 5
2.4 MAC Generation Algorithm 7
2.5 MAC Verification Algorithm 9
3. Security Considerations 10
4. Test Vector 11
5. Acknowledgement 11
6. Authors address 12
7. References 13
Appendix A. Test Code 14

1. Introduction

National Institute of Standards and Technology (NIST) has newly specified the Cipher-based Message Authentication Code (CMAC). CMAC [NIST-CMAC] is a keyed hash function that is based on a symmetric key block cipher such as the Advanced Encryption Standard [NIST-AES]. CMAC is equivalent to the One-Key CBC MAC1 (OMAC1) submitted by Iwata and Kurosawa [OMAC1a, OMAC1b]. OMAC1 is an improvement of the eXtended Cipher Block Chaining mode (XCBC) submitted by Black and Rogaway [XCBCa, XCBCb], which itself is an improvement of the basic CBC-MAC. XCBC efficiently addresses the security deficiencies of CBC-MAC, and OMAC1 efficiently reduces the key size of XCBC.

AES-CMAC provides stronger assurance of data integrity than a checksum or an error detecting code. The verification of a checksum or an error detecting code detects only accidental modifications of the data, while CMAC is designed to detect intentional, unauthorized modifications of the data, as well as accidental modifications.

AES-CMAC achieves the similar security goal of HMAC [RFC-HMAC]. Since AES-CMAC is based on a symmetric key block cipher, AES, while HMAC is based on a hash function, such as SHA-1, AES-CMAC is appropriate for information systems in which AES is more readily available than a hash function.

This memo specifies the authentication algorithm based on CMAC with AES-128. This new authentication algorithm is named AES-CMAC.

2. Specification of AES-CMAC

2.1 Basic definitions

The following table describes the basic definitions necessary to explain the specification of AES-CMAC.

```
x || y          Concatenation.
                x || y is the string x concatenated with string y.
                0000 || 1111 is 00001111.

x XOR y         Exclusive-OR operation.
                For two equal length strings x and y,
                x XOR y is their bit-wise exclusive-OR.

ceil(x)         Ceiling function.
                The smallest integer no smaller than x.
                ceil(3.5) is 4. ceil(5) is 5.

x << 1          Left-shift of the string x by 1 bit.
                The most significant bit disappears and a zero
                comes into the least significant bit.
                10010001 << 1 is 00100010.

0^n             The string that consists of n zero-bits.
                0^3 means that 000 in binary format.
                10^4 means that 10000 in binary format.
                10^i means that 1 followed by i-times repeated
                zero's.

MSB(x)          The most-significant bit of the string x.
                MSB(10010000) means 1.

padding(x)      10^i padded output of input x.
                It is described in detail in section 2.4.

Key             128 bits (16 bytes) long key for AES-128.
                Denoted by K.

First subkey    128 bits (16 bytes) long first subkey,
                derived through the subkey
                generation algorithm from the key K.
                Denoted by K1.

Second subkey   128 bits (16 bytes) long second subkey,
                derived through the subkey
                generation algorithm from the key K.
                Denoted by K2.
```

```
Message          A message to be authenticated.
                 Denoted by M.
                 The message can be null, which means that the length
                 of M is 0.

Message length   The length of the message M in bytes.
                 Denoted by len.
                 Minimum value of the length can be 0.  The maximum
                 value of the length is not specified in this document.

AES-128(K,M)     AES-128(K,M) is the 128-bit ciphertext of AES-128
                 for a 128-bit key K and a 128-bit message M.

MAC              A 128-bit string which is the output of AES-CMAC.
                 Denoted by T.
                 Validating the MAC provides assurance of the
                 integrity and authenticity over the message from
                 the source.

MAC length       By default, the length of the output of AES-CMAC is
                 128 bits. It is possible to truncate the MAC.
                 Result of truncation should be taken in most
                 significant bits first order. MAC length must be
                 specified before the communication starts, and
                 it must not be changed during the life time of the key.
```

2.2 Overview

AES-CMAC uses the Advanced Encryption Standard [NIST-AES] as a building block. To generate a MAC, AES-CMAC takes a secret key, a message of variable length and the length of the message in bytes as inputs, and returns a fixed bit string called a MAC.

The core of AES-CMAC is the basic CBC-MAC. For a message M to be authenticated, the CBC-MAC is applied to M. There are two cases of operation in CMAC. Figure 2.1 illustrated the operation of CBC-MAC with two cases. If the size of input message block is equal to multiple of block size namely 128 bits, the last block processing shall be exclusive-OR'ed with K1. Otherwise, the last block shall be padded with 10^i (notation is described in section 2.1) and exclusive-OR'ed with K2. The result of the previous process will be the input of the last CBC operation. The output of AES-CMAC provides data integrity over whole input message.

```
+-----+     +-----+     +-----+     +-----+     +-----+     +---+----+
| M_1 |     | M_2 |     | M_n |     | M_1 |     | M_2 |     |M_n|10^i|
+-----+     +-----+     +-----+     +-----+     +-----+     +---+----+
   |           |           |   +--+    |           |           |   +--+
   |     +--->(+)    +--->(+)<-|K1|    |     +--->(+)    +--->(+)<-|K2|
   |     |     |     |     |   +--+    |     |     |     |     |   +--+
+-----+  |  +-----+  |  +-----+     +-----+  |  +-----+  |  +-----+
|AES_K|  |  |AES_K|  |  |AES_K|     |AES_K|  |  |AES_K|  |  |AES_K|
+-----+  |  +-----+  |  +-----+     +-----+  |  +-----+  |  +-----+
   |     |     |     |     |           |     |     |     |     |
   +-----+     +-----+     |           +-----+     +-----+     |
                           |                                   |
                        +-----+                              +-----+
                        |  T  |                              |  T  |
                        +-----+                              +-----+

            (a) positive multiple block length         (b) otherwise

             Figure 2.1 Illustration of two cases of AES-CMAC.
```

AES_K is AES-128 with key K.
The message M is divided into blocks M_1,...,M_n,
where M_i is the i-th message block.
The length of M_i is 128 bits for i = 1,...,n-1, and
the length of the last block M_n is less than or equal to 128 bits.
K1 is the subkey for the case (a), and
K2 is the subkey for the case (b).
K1 and K2 are generated by the subkey generation algorithm
described in section 2.3.

2.3 Subkey Generation Algorithm

The subkey generation algorithm, Generate_Subkey(), takes a secret key, K, which is just the key for AES-128.

The output of the subkey generation algorithm is two subkeys, K1 and K2. We write (K1,K2) := Generate_Subkey(K).

Subkeys K1 and K2 are used in both MAC generation and MAC verification algorithms. K1 is used for the case where the length of the last block is equal to the block length. K2 is used for the case where the length of the last block is less than the block length.

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                    Algorithm Generate_Subkey                      +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                                                                   +
+   Input    : K (128-bit key)                                      +
+   Output   : K1 (128-bit first subkey)                            +
+              K2 (128-bit second subkey)                           +
+-------------------------------------------------------------------+
+                                                                   +
+   Constants: const_Zero is 0x00000000000000000000000000000000     +
+              const_Rb   is 0x00000000000000000000000000000087     +
+   Variables: L          for output of AES-128 applied to 0^128    +
+                                                                   +
+   Step 1.  L := AES-128(K, const_Zero);                           +
+   Step 2.  if MSB(L) is equal to 0                                +
+            then    K1 := L << 1;                                  +
+            else    K1 := (L << 1) XOR const_Rb;                   +
+   Step 3.  if MSB(K1) is equal to 0                               +
+            then    K2 := K1 << 1;                                 +
+            else    K2 := (K1 << 1) XOR const_Rb;                  +
+   Step 4.  return K1, K2;                                         +
+                                                                   +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 2.2 Algorithm Generate_Subkey

Figure 2.2 specifies the subkey generation algorithm.
In step 1, AES-128 is applied to all zero bits with the input key K.

In step 2, K1 is derived through the following operation:
If the most significant bit of L is equal to 0, K1 is the left-shift of L by 1-bit.
Otherwise, K1 is the exclusive-OR of const_Rb and the left-shift of L by 1-bit.

In step 3, K2 is derived through the following operation:
If the most significant bit of K1 is equal to 0, K2 is the left-shift of K1 by 1-bit.
Otherwise, K2 is the exclusive-OR of const_Rb and the left-shift of K1 by 1-bit.

In step 4, (K1,K2) := Generate_Subkey(K) is returned.

The mathematical meaning of procedure in step 2 and step 3 including const_Rb can be found in [OMAC1a].

2.4 MAC Generation Algorithm

The MAC generation algorithm, AES-CMAC(), takes three inputs, a secret key, a message, and the length of the message in bytes. The secret key, denoted by K, is just the key for AES-128. The message and its length in bytes are denoted by M and len, respectively. The message M is denoted by the sequence of M_i where M_i is the i-th message block. That is, if M consists of n blocks, then M is written as

- M = M_1 || M_2 || ... || M_{n-1} || M_n

The length of M_i is 128 bits for i = 1,...,n-1, and the length of the last block M_n is less than or equal to 128 bits.

The output of the MAC generation algorithm is a 128-bit string, called a MAC, which can be used to validate the input message. The MAC is denoted by T and we write T := AES-CMAC(K,M,len). Validating the MAC provides assurance of the integrity and authenticity over the message from the source.

It is possible to truncate the MAC. According to [NIST-CMAC] at least 64-bit MAC should be used for against guessing attack. Result of truncation should be taken in most significant bits first order.

The block length of AES-128 is 128 bits (16 bytes). There is a special treatment in case that the length of the message is not a positive multiple of the block length. The special treatment is to pad 10^i bit-string for adjusting the length of the last block up to the block length.

For the input string x of r-bytes, where r < 16, the padding function, padding(x), is defined as follows.

- padding(x) = x || 10^i where i is 128-8*r-1

That is, padding(x) is the concatenation of x and a single '1' followed by the minimum number of '0's so that the total length is equal to 128 bits.

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                    Algorithm AES-CMAC                             +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                                                                   +
+   Input    : K    ( 128-bit key )                                 +
+            : M    ( message to be authenticated )                 +
+            : len  ( length of the message in bytes )              +
+   Output   : T    ( message authenticated code )                  +
+                                                                   +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+   Constants: const_Zero is 0x00000000000000000000000000000000     +
+              const_Rb   is 0x00000000000000000000000000000087     +
+              const_Bsize is 16                                    +
+                                                                   +
+   Variables: K1, K2 for 128-bit subkeys                           +
+              M_i is the i-th block (i=1..ceil(len/const_Bsize))   +
+              M_last is the last block xor-ed with K1 or K2        +
+              n      for number of blocks to be processed          +
+              r      for number of bytes of last block             +
+              flag   for denoting if last block is complete or not +
+                                                                   +
+   Step 1.  (K1,K2) := Generate_Subkey(K);                         +
+   Step 2.  n := ceil(len/const_Bsize);                            +
+   Step 3.  if n = 0                                               +
+            then                                                   +
+                 n := 1;                                           +
+                 flag := false;                                    +
+            else                                                   +
+                 if len mod const_Bsize is 0                       +
+                 then flag := true;                                +
+                 else flag := false;                               +
+                                                                   +
+   Step 4.  if flag is true                                        +
+            then M_last := M_n XOR K1;                             +
+            else M_last := padding(M_n) XOR K2;                    +
+   Step 5.  X := const_Zero;                                       +
+   Step 6.  for i := 1 to n-1 do                                   +
+                begin                                              +
+                  Y := X XOR M_i;                                  +
+                  X := AES-128(K,Y);                               +
+                end                                                +
+            Y := M_last XOR X;                                     +
+            T := AES-128(K,Y);                                     +
+   Step 7.  return T;                                              +
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 2.3 Algorithm AES-CMAC

Figure 2.3 describes the MAC generation algorithm.

In step 1, subkeys K1 and K2 are derived from K through the subkey generation algorithm.

In step 2, the number of blocks, n, is calculated. The number of blocks is the smallest integer value greater than or equal to quotient by dividing length parameter by the block length, 16 bytes.

In step 3, the length of the input message is checked. If the input length is less than 128 bits (including null), the number of blocks to be processed shall be 1 and mark the flag as not-complete-block (false). Otherwise, if the last block length is 128 bits, mark the flag as complete-block (true), else mark the flag as not-complete-block (false).

In step 4, M_last is calculated by exclusive-OR'ing M_n and previously calculated subkeys. If the last block is a complete block (true), then M_last is the exclusive-OR of M_n and K1. Otherwise, M_last is the exclusive-OR of padding(M_n) and K2.

In step 5, the variable X is initialized.

In step 6, the basic CBC-MAC is applied to M_1,...,M_{n-1},M_last.

In step 7, the 128-bit MAC, T := AES-CMAC(K,M,len), is returned.

If necessary, truncation of the MAC is done before returning the MAC.

2.5 MAC Verification Algorithm

The verification of the MAC is simply done by a MAC recomputation. We use the MAC generation algorithm which is described in section 2.4.

The MAC verification algorithm, Verify_MAC(), takes four inputs, a secret key, a message, the length of the message in bytes, and the received MAC. They are denoted by K, M, len, and T' respectively.

The output of the MAC verification algorithm is either INVALID or VALID.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                      Algorithm Verify_MAC                          +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+                                                                    +
+   Input    : K    ( 128-bit Key )                                  +
+            : M    ( message to be verified )                       +
+            : len  ( length of the message in bytes )               +
+            : T'   ( the received MAC to be verified )              +
+   Output   : INVALID or VALID                                      +
+                                                                    +
+-------------------------------------------------------------------+
+                                                                    +
+   Step 1.  T* := AES-CMAC(K,M,len);                                +
+   Step 2.  if T* = T'                                              +
+            then                                                    +
+                 return VALID;                                      +
+            else                                                    +
+                 return INVALID;                                    +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 2.4 Algorithm Verify_MAC

Figure 2.4 describes the MAC verification algorithm.

In step 1, T* is derived from K, M and len through the MAC generation algorithm.

In step 2, T* and T' are compared. If T*=T', then return VALID, otherwise return INVALID.

If the output is INVALID, then the message is definitely not authentic, i.e., it did not originate from a source that executed the generation process on the message to produce the purported MAC.

If the output is VALID, then the design of the AES-CMAC provides assurance that the message is authentic and, hence, was not corrupted in transit; however, this assurance, as for any MAC algorithm, is not absolute.

3. Security Considerations

The security provided by AES-CMAC are built on strong cryptographic algorithm AES. However as is true with any cryptographic algorithm, part of its strength lies in the secret key, 'K' and the correctness of the implementation in all of the participating systems.
If the secret key 'K' is compromised or inappropriately shared, it no longer guarantee either authentication or integrity of message.
The secret key shall be generated in a way that meet the pseudo randomness requirement of RFC 4086 [RFC4086] and should be kept in safe. If and only if AES-CMAC used properly it can provide the Authentication and Integrity that meet the best current practice

of message authentication.

4. Test Vectors

Following test vectors are the same as those of [NIST-CMAC].
The following vectors are also output of the test program in
appendix A.

```
--------------------------------------------------
Subkey Generation
K              2b7e1516 28aed2a6 abf71588 09cf4f3c
AES-128(key,0) 7df76b0c 1ab899b3 3e42f047 b91b546f
K1             fbeed618 35713366 7c85e08f 7236a8de
K2             f7ddac30 6ae266cc f90bc11e e46d513b
--------------------------------------------------

--------------------------------------------------
Example 1: len = 0
M              <empty string>
AES-CMAC       bb1d6929 e9593728 7fa37d12 9b756746
--------------------------------------------------

Example 2: len = 16
M              6bc1bee2 2e409f96 e93d7e11 7393172a
AES-CMAC       070a16b4 6b4d4144 f79bdd9d d04a287c
--------------------------------------------------

Example 3: len = 40
M              6bc1bee2 2e409f96 e93d7e11 7393172a
               ae2d8a57 1e03ac9c 9eb76fac 45af8e51
               30c81c46 a35ce411
AES-CMAC       dfa66747 de9ae630 30ca3261 1497c827
--------------------------------------------------

Example 4: len = 64
M              6bc1bee2 2e409f96 e93d7e11 7393172a
               ae2d8a57 1e03ac9c 9eb76fac 45af8e51
               30c81c46 a35ce411 e5fbc119 1a0a52ef
               f69f2445 df4f9b17 ad2b417b e66c3710
AES-CMAC       51f0bebf 7e3b9d92 fc497417 79363cfe
--------------------------------------------------
```

5. Acknowledgement

Portions of this text here in is borrowed from [NIST-CMAC].
We appreciate OMAC1 authors and SP 800-38B author, and Russ Housley
for his useful comments and guidance that have been incorporated
herein. We also appreciate David Johnston for providing code of
the AES block cipher.

6. Author's Address

Junhyuk Song
University of Washington
Samsung Electronics
(206) 853-5843
songlee@ee.washington.edu
junhyuk.song@samsung.com

Jicheol Lee
Samsung Electronics
+82-31-279-3605
jicheol.lee@samsung.com

Radha Poovendran
Network Security Lab
University of Washington
(206) 221-6512
radha@ee.washington.edu

Tetsu Iwata
Ibaraki University
iwata@cis.ibaraki.ac.jp

7. References

7.1. Normative References

[NIST-CMAC] NIST, Special Publication 800-38B Draft,"Recommendation for Block Cipher Modes of Operation: The CMAC Method for Authentication," March 9, 2005

[NIST-AES] NIST, FIPS 197, "Advanced Encryption Standard (AES)," November 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[OMAC1] "OMAC: One-Key CBC MAC," Tetsu Iwata and Kaoru Kurosawa, Department of Computer and Information Sciences, Ilbaraki University, March 10, 2003.

[XCBC] Black, J. and P. Rogaway, "A Suggestion for Handling Arbitrary-Length Messages with the CBC MAC," NIST Second Modes of Operation Workshop, August 2001. http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/xcbc-mac/xcbc-mac-spec.pdf

[RFC4086] Eastlake 3rd, D., Crocker, S., and J. Schiller, "Randomness Requirements for Security", RFC 4086 June 2005

7.2. Informative References

[OMAC1a] Tetsu Iwata and Kaoru Kurosawa, "OMAC: One-Key CBC MAC," Fast Software Encryption, FSE 2003, LNCS 2887, pp. 129-153, Springer-Verlag, 2003.

[RFC-HMAC] Hugo Krawczyk, Mihir Bellare and Ran Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC2104, February 1997.

[OMAC1b] Tetsu Iwata and Kaoru Kurosawa, "OMAC: One-Key CBC MAC," Submission to NIST, December 2002. Available from the NIST modes of operation web site at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/omac/omac-spec.pdf

[XCBCa] John Black and Phillip Rogaway, "A Suggestion for Handling Arbitrary-Length Messages with the CBC MAC," NIST Second Modes of Operation Workshop, August 2001. Available from the NIST modes of operation web site at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/xcbc-mac/xcbc-mac-spec.pdf

[XCBCb] John Black and Phillip Rogaway, "CBC MACs for Arbitrary-Length Messages: The Three-Key Constructions," Journal of Cryptology, Vol. 18, No. 2, pp. 111-132, Springer-Verlag, Spring 2005.

[RFC1750] Eastlake 3rd, D., Crocker, S., and J. Schiller, "Randomness Recommendations for Security", RFC 1750, December 1994.

Appendix A. Test Code

```
/****************************************************************/
/* AES-CMAC with AES-128 bit                                    */
/* AES-128 from David Johnston (802.16)                         */
/* CMAC     Algorithm described in SP800-38B draft              */
/* Author: Junhyuk Song (junhyuk.song@samsung.com)              */
/*         Jicheol Lee  (jicheol.lee@samsung.com)               */
/****************************************************************/

#include <stdio.h>

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

```
/* For CMAC Calculation */
unsigned char const_Rb[16] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x87
};
unsigned char const_Zero[16] = {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

/*****************************/
/**** Function Prototypes ****/
/*****************************/

void xor_128(unsigned char *a, unsigned char *b, unsigned char *out);
void xor_32(unsigned char *a, unsigned char *b, unsigned char *out);
unsigned char sbox(unsigned char a);
void next_key(unsigned char *key, int round);
void byte_sub(unsigned char *in, unsigned char *out);
void shift_row(unsigned char *in, unsigned char *out);
void mix_column(unsigned char *in, unsigned char *out);
void add_round_key( unsigned char *shiftrow_in,
                    unsigned char *mcol_in,
                    unsigned char *block_in,
                    int round,
                    unsigned char *out);

void AES_128(unsigned char *key, unsigned char *data, unsigned char
             *ciphertext);
void leftshift_onebit(unsigned char *input,unsigned char *output);


/****************************************/
/* AES_128()                            */
/* Performs a 128 bit AES encrypt with  */
/* 128 bit data.                        */
/****************************************/

void xor_128(unsigned char *a, unsigned char *b, unsigned char *out)
{
    int i;
    for (i=0;i<16; i++)
    {
        out[i] = a[i] ^ b[i];
    }
}
```

```
void xor_32(unsigned char *a, unsigned char *b, unsigned char *out)
{
    int i;
    for (i=0;i<4; i++)
    {
        out[i] = a[i] ^ b[i];
    }
}

unsigned char sbox(unsigned char a)
{
    return sbox_table[(int)a];
}

void next_key(unsigned char *key, int round)
{
    unsigned char rcon;
    unsigned char sbox_key[4];
    unsigned char rcon_table[12] = {
        0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
        0x1b, 0x36, 0x36, 0x36
    };

    sbox_key[0] = sbox(key[13]);
    sbox_key[1] = sbox(key[14]);
    sbox_key[2] = sbox(key[15]);
    sbox_key[3] = sbox(key[12]);
    rcon = rcon_table[round];

    xor_32(&key[0], sbox_key, &key[0]);
    key[0] = key[0] ^ rcon;

    xor_32(&key[4], &key[0], &key[4]);
    xor_32(&key[8], &key[4], &key[8]);
    xor_32(&key[12], &key[8], &key[12]);
}

void byte_sub(unsigned char *in, unsigned char *out)
{
    int i;
    for (i=0; i< 16; i++)
    {
        out[i] = sbox(in[i]);
    }
}
```

```
void shift_row(unsigned char *in, unsigned char *out)
{
    out[0] =  in[0];
    out[1] =  in[5];
    out[2] =  in[10];
    out[3] =  in[15];
    out[4] =  in[4];
    out[5] =  in[9];
    out[6] =  in[14];
    out[7] =  in[3];
    out[8] =  in[8];
    out[9] =  in[13];
    out[10] = in[2];
    out[11] = in[7];
    out[12] = in[12];
    out[13] = in[1];
    out[14] = in[6];
    out[15] = in[11];
}

void mix_column(unsigned char *in, unsigned char *out)
{
    int i;
    unsigned char add1b[4];
    unsigned char add1bf7[4];
    unsigned char rotl[4];
    unsigned char swap_halfs[4];
    unsigned char andf7[4];
    unsigned char rotr[4];
    unsigned char temp[4];
    unsigned char tempb[4];

    for (i=0 ; i<4; i++)
    {
        if ((in[i] & 0x80)== 0x80)
            add1b[i] = 0x1b;
        else
            add1b[i] = 0x00;
    }


    swap_halfs[0] = in[2];    /* Swap halfs */
    swap_halfs[1] = in[3];
    swap_halfs[2] = in[0];
    swap_halfs[3] = in[1];

    rotl[0] = in[3];        /* Rotate left 8 bits */
    rotl[1] = in[0];
    rotl[2] = in[1];
```

```
   rotl[3] = in[2];
```

```
    andf7[0] = in[0] & 0x7f;
    andf7[1] = in[1] & 0x7f;
    andf7[2] = in[2] & 0x7f;
    andf7[3] = in[3] & 0x7f;

    for (i = 3; i>0; i--)    /* logical shift left 1 bit */
    {
        andf7[i] = andf7[i] << 1;
        if ((andf7[i-1] & 0x80) == 0x80)
        {
            andf7[i] = (andf7[i] | 0x01);
        }
    }
    andf7[0] = andf7[0] << 1;
    andf7[0] = andf7[0] & 0xfe;

    xor_32(add1b, andf7, add1bf7);

    xor_32(in, add1bf7, rotr);

    temp[0] = rotr[0];         /* Rotate right 8 bits */
    rotr[0] = rotr[1];
    rotr[1] = rotr[2];
    rotr[2] = rotr[3];
    rotr[3] = temp[0];
    xor_32(add1bf7, rotr, temp);
    xor_32(swap_halfs, rotl,tempb);
    xor_32(temp, tempb, out);
}

void AES_128(unsigned char *key, unsigned char *data, unsigned char
*ciphertext)
{
    int round;
    int i;
    unsigned char intermediatea[16];
    unsigned char intermediateb[16];
    unsigned char round_key[16];

    for(i=0; i<16; i++) round_key[i] = key[i];
    for (round = 0; round < 11; round++)
    {
        if (round == 0)
        {
            xor_128(round_key, data, ciphertext);
            next_key(round_key, round);
        }
```

```
        else if (round == 10)
        {
            byte_sub(ciphertext, intermediatea);
            shift_row(intermediatea, intermediateb);
            xor_128(intermediateb, round_key, ciphertext);
        }
        else    /* 1 - 9 */
        {
            byte_sub(ciphertext, intermediatea);
            shift_row(intermediatea, intermediateb);
            mix_column(&intermediateb[0], &intermediatea[0]);
            mix_column(&intermediateb[4], &intermediatea[4]);
            mix_column(&intermediateb[8], &intermediatea[8]);
            mix_column(&intermediateb[12], &intermediatea[12]);
            xor_128(intermediatea, round_key, ciphertext);
            next_key(round_key, round);
        }
    }
}


void print_hex(char *str, unsigned char *buf, int len)
{
    int     i;

    for ( i=0; i<len; i++ ) {
        if ( (i % 16) == 0 && i != 0 ) printf(str);
        printf("%02x", buf[i]);
        if ( (i % 4) == 3 ) printf(" ");
        if ( (i % 16) == 15 ) printf("\n");
    }
    if ( (i % 16) != 0 ) printf("\n");
}

void print128(unsigned char *bytes)
{
    int         j;
    for (j=0; j<16;j++) {
        printf("%02x",bytes[j]);
        if ( (j%4) == 3 ) printf(" ");
    }
}

void print96(unsigned char *bytes)
{
    int         j;
    for (j=0; j<12;j++) {
        printf("%02x",bytes[j]);
        if ( (j%4) == 3 ) printf(" ");
```

```
    }
}
```

```
/* AES-CMAC Generation Function */

void leftshift_onebit(unsigned char *input,unsigned char *output)
{
    int         i;
    unsigned char overflow = 0;

    for ( i=15; i>=0; i-- ) {
        output[i] = input[i] << 1;
        output[i] |= overflow;
        overflow = (input[i] & 0x80)?1:0;
    }
    return;
}

void generate_subkey(unsigned char *key, unsigned char *K1, unsigned
                     char *K2)
{
    unsigned char L[16];
    unsigned char Z[16];
    unsigned char tmp[16];
    int i;

    for ( i=0; i<16; i++ ) Z[i] = 0;

    AES_128(key,Z,L);

    if ( (L[0] & 0x80) == 0 ) { /* If MSB(L) = 0, then K1 = L << 1 */
        leftshift_onebit(L,K1);
    } else {    /* Else K1 = ( L << 1 ) (+) Rb */
        leftshift_onebit(L,tmp);
        xor_128(tmp,const_Rb,K1);
    }

    if ( (K1[0] & 0x80) == 0 ) {
        leftshift_onebit(K1,K2);
    } else {
        leftshift_onebit(K1,tmp);
        xor_128(tmp,const_Rb,K2);
    }
    return;
}
```

```
void padding ( unsigned char *lastb, unsigned char *pad, int length )
{
    int         j;

    /* original last block */
    for ( j=0; j<16; j++ ) {
        if ( j < length ) {
            pad[j] = lastb[j];
        } else if ( j == length ) {
            pad[j] = 0x80;
        } else {
            pad[j] = 0x00;
        }
    }
}

void AES_CMAC ( unsigned char *key, unsigned char *input, int length,
                unsigned char *mac )
{
    unsigned char       X[16],Y[16], M_last[16], padded[16];
    unsigned char       K1[16], K2[16];
    int         n, i, flag;
    generate_subkey(key,K1,K2);

    n = (length+15) / 16;       /* n is number of rounds */

    if ( n == 0 ) {
        n = 1;
        flag = 0;
    } else {
        if ( (length%16) == 0 ) { /* last block is a complete block */
            flag = 1;
        } else { /* last block is not complete block */
            flag = 0;
        }
    }

    if ( flag ) { /* last block is complete block */
        xor_128(&input[16*(n-1)],K1,M_last);
    } else {
        padding(&input[16*(n-1)],padded,length%16);
        xor_128(padded,K2,M_last);
    }

    for ( i=0; i<16; i++ ) X[i] = 0;
    for ( i=0; i<n-1; i++ ) {
        xor_128(X,&input[16*i],Y); /* Y := Mi (+) X  */
        AES_128(key,Y,X);      /* X := AES-128(KEY, Y); */
    }
```

```
    xor_128(X,M_last,Y);
    AES_128(key,Y,X);

    for ( i=0; i<16; i++ ) {
        mac[i] = X[i];
    }
}

int main()
{
    unsigned char L[16], K1[16], K2[16], T[16], TT[12];
    unsigned char M[64] = {
        0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
        0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
        0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
        0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51,
        0x30, 0xc8, 0x1c, 0x46, 0xa3, 0x5c, 0xe4, 0x11,
        0xe5, 0xfb, 0xc1, 0x19, 0x1a, 0x0a, 0x52, 0xef,
        0xf6, 0x9f, 0x24, 0x45, 0xdf, 0x4f, 0x9b, 0x17,
        0xad, 0x2b, 0x41, 0x7b, 0xe6, 0x6c, 0x37, 0x10
    };
    unsigned char key[16] = {
        0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
        0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c
    };

    printf("--------------------------------------------------\n");
    printf("K              "); print128(key); printf("\n");

    printf("\nSubkey Generation\n");
    AES_128(key,const_Zero,L);
    printf("AES_128(key,0) "); print128(L); printf("\n");
    generate_subkey(key,K1,K2);
    printf("K1             "); print128(K1); printf("\n");
    printf("K2             "); print128(K2); printf("\n");

    printf("\nExample 1: len = 0\n");
    printf("M              "); printf("<empty string>\n");

    AES_CMAC(key,M,0,T);
    printf("AES_CMAC       "); print128(T); printf("\n");

    printf("\nExample 2: len = 16\n");
    printf("M              "); print_hex("                ",M,16);
    AES_CMAC(key,M,16,T);
    printf("AES_CMAC       "); print128(T); printf("\n");
    printf("\nExample 3: len = 40\n");
    printf("M              "); print_hex("               ",M,40);
    AES_CMAC(key,M,40,T);
```

```
    printf("AES_CMAC       "); print128(T); printf("\n");
```

```
    printf("\nExample 4: len = 64\n");
    printf("M              "); print_hex("               ",M,64);
    AES_CMAC(key,M,64,T);
    printf("AES_CMAC       "); print128(T); printf("\n");

    printf("--------------------------------------------------\n");

    return 0;
}
```

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

Copyright Statement

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.