IEEE P1619.2™/D9
Draft Standard for Wide-Block Encryption for Shared Storage Media

Prepared by the Security in Storage Working Group of the

Computer Society Information Assurance Committee

and

Storage Systems Standards Committee

of the

IEEE Computer Society

Three Park Avenue
New York, New York 10016-5997, USA

This document is an unapproved draft of a proposed IEEE Standard. As such, this document is subject to change. USE AT YOUR OWN RISK! Because this is an unapproved draft, this document must not be utilized for any conformance/compliance purposes. Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of international standardization consideration. Prior to adoption of this document, in whole or in part, by another standards development organization, permission must first be obtained from the IEEE Standards Activities Department (stds.ipr@ieee.org). Other entities seeking permission to reproduce this document, in whole or in part, must also obtain permission from the IEEE Standards Activities Department.

IEEE Standards Activities Department
445 Hoes Lane
Piscataway, NJ 08854, USA

Abstract: This standard specifies an architecture for encryption of data in random access storage devices, oriented towards applications which benefit from wide encryption-block sizes of 512 bytes and above.

Keywords: data-at-rest security, encryption, security, storage, extended codebook mode of operation (XCB), encrypt-mix-encrypt-v2 mode of operation (EME2), encryption with associated data (EAD)

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

PDF: ISBN 978-0-XXXX-XXXX-X STDXXXX
Print: ISBN 978-0-XXXX-XXXX-X STDPDXXXX

This page is left blank intentionally.

Introduction

This introduction is not part of IEEE P<designation>/D<draft_number>, Draft Standard for Wide-Block Encryption for Shared Storage Media.

The purpose of this standard, similarly to IEEE-1619-2007, is to describe a method of encryption for data stored in sector-based devices, where the threat model includes possible access to stored data by the adversary. As in IEEE-1619-2007, this standard specifies length-preserving encryption transforms to be applied to the plaintext sector before storing it on the storage media.

This standard improves on IEEE-1619-2007, by defining “wide block” encryption transforms. This means that they act on the whole sector at once, and each bit on the input plaintext influences every bit of the output ciphertext (and vise-versa for decryption). In particular, this standard specifies the EME2-AES and the XCB-AES wide-block encryption transforms.

Wide-block encryption better hides plaintext statistics, and provides better protection than the narrow-block encryption defined in IEEE-1619-2007 against attacks that involve traffic analysis and/or manipulations of ciphertext on the ~~raw~~ storage media.

Notice to users

Laws and regulations

Users of these documents should consult all applicable laws and regulations. Compliance with the provisions of this standard does not imply compliance to any applicable regulatory requirements. Implementers of the standard are responsible for observing or referring to the applicable regulatory requirements. IEEE does not, by the publication of its standards, intend to urge action that is not in compliance with applicable laws, and these documents may not be construed as doing so.

Copyrights

This document is copyrighted by the IEEE. It is made available for a wide variety of both public and private uses. These include both use, by reference, in laws and regulations, and use in private self-regulation, standardization, and the promotion of engineering practices and methods. By making this document available for use and adoption by public authorities and private users, the IEEE does not waive any rights in copyright to this document.

Updating of IEEE documents

Users of IEEE standards should be aware that these documents may be superseded at any time by the issuance of new editions or may be amended from time to time through the issuance of amendments, corrigenda, or errata. An official IEEE document at any point in time consists of the current edition of the document together with any amendments, corrigenda, or errata then in effect. In order to determine whether a given document is the current edition and whether it has been amended through the issuance of amendments, corrigenda, or errata, visit the IEEE Standards Association web site at http://ieeexplore.ieee.org/xpl/standards.jsp, or contact the IEEE at the address listed previously.

For more information about the IEEE Standards Association or the IEEE standards development process, visit the IEEE-SA web site at http://standards.ieee.org.

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

Patents

Attention is called to the possibility that implementation of this Standard may require use of subject matter covered by patent rights. By publication of this Standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE is not responsible for identifying Essential Patent Claims for which a license may be required, for conducting inquiries into the legal validity or scope of Patents Claims or determining whether any licensing terms or conditions provided in connection with submission of a Letter of Assurance, if any, or in any licensing agreements are reasonable or non-discriminatory. Users of this Standard are expressly advised that determination of the validity of any patent rights, and the risk of infringement of such rights, is entirely their own responsibility. Further information may be obtained from the IEEE Standards Association.

Participants

At the time this draft standard was submitted to the IEEE-SA Standards Board for approval, the Security in Storage Working Group operated under the following sponsorship:

John L. Cole, *Sponsor*

Curtis Anderson, *Co-Sponsor*

At the time this draft standard was submitted to the IEEE-SA Standards Board for approval, the Security in Storage Working Group had the following officers:

Matthew V. Ball, *Chair*

Eric Hibbard, *Vice-Chair*

Fabio Maino, *Secretary*

At the time this draft Standard was submitted to the IEEE-SA Standards Board for approval, the 1619.2 Task Group had the following membership:

James Hughes, *Chair*

Eric Hibbard, *Vice-Chair*

Fabio Maino, *Technical Editor*

Gideon Avida
Matthew V. Ball
Jim Coomes
Boris Dolgunov
Rob Elliott
Hal Finney
John Geldman
Bob Griffin
Cyril Guyot
Shai Halevi
Laszlo Hars
Eric Hibbard
Larry Hofer
Walt Hubis
James Hughes
Glen Jaquette
Scott Kipp
Curt Kolovson
Bob Lockhart
Fabio Maino
Charlie Martin
David McGrew
Gary Moorhead
Bob Nixon
Landon Noll
Jim Norton
Scott Painter
Dave Peterson
Serge Plotkin
Niels Reimers
Subhash Sankuratripati
David Sheehy
Bob Snively
Joel Spencer
Doug Whiting
Mike Witkowski

Special Thanks to: Hal Finney, Brian Gladman, Shai Halevi, David McGrew.

The following members of the **[individual/entity]** balloting committee voted on this Standard. Balloters may have voted for approval, disapproval, or abstention.

(to be supplied by IEEE)

CONTENTS

<After draft body is complete, select this text and click Insert Special->Add (Table of) Contents>

Draft Standard for Wide-Block Encryption for Shared Storage Media

1. Overview

1.1 Scope

This standard specifies an architecture for encryption of data in random access storage devices, oriented toward applications which benefit from wide encryption-block sizes of 512 bytes and above.

1.2 Purpose

This standard specifies an architecture for media security and enabling components. Wide encryption blocks are well suited to environments where the attacker has repeated access to cryptographic communication or ciphertext, or is able to perform traffic analysis of data access patterns. The standard is oriented toward fixed-size encryption blocks without data ~~expansion, but anticipates an optional data expansion mode to resist attacks involving data tampering.~~

2. Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments and corrigenda) applies.

[N1] IEEE Std 1619™, IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices.[1,2]

[N2] NIST FIPS 197, Federal Information Processing Standard (FIPS) 197 (November 26, 2001), Announcing the Advanced Encryption Standard (AES).

---

[1] IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

[2] The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[N3] NIST Special Publication 800-38A (NIST SP 800-38A), Recommendation for Block Cipher Modes of Operation—Methods and Techniques.

3. Definitions, Acronyms, and Abbreviations

3.1 Keywords

For the purposes of this standard, the following terms are keywords.

can: A keyword indicating a capability (*can* equals *is able to*).

required: See **shall**.

may: A keyword indicating a course of action permissible within the limits of this standard (*may* equals *is permitted to*).

must: A keyword used only to describe an unavoidable situation that does not constitute a requirement for compliance to this standard.

shall: A keyword indicating a mandatory requirement strictly to be followed in order to conform to this standard and from which no deviation is permitted (*shall* equals *is required to*).

shall not: A phrase indicating an absolute prohibition of this standard.

should: A keyword indicating that among several possibilities one is recommended as particularly suitable, without mentioning or excluding others; or that a certain course of action is preferred but not necessarily required; or that (in the negative form) a certain course of action is deprecated but not prohibited (*should* equals *is recommended to*).

3.2 Definitions

For the purposes of this draft standard, the following terms and definitions apply. The Authoritative Dictionary of IEEE Standards, Seventh Edition, should be referenced for terms not defined in this clause.

associated data: Data that is associated with the plaintext, but which does not need to be encrypted. The associated data input should characterize the plaintext, and it should be as fine-grained as possible. In other algorithms for encryption of data at rest, the associated data is often referred as “tweak value”.

encryption with associated data (EAD): A cryptographic algorithm that consists of an encrypt used to encrypt a plaintext and the associated data with a secret key, and a decrypt procedure used to decrypt a ciphertext and the associate encrypted data with the same secret key.

3.3 Acronyms and Abbreviations

AES	Advanced Encryption Standard
FIPS	Federal Information Processing Standard

GF	Galois Field (see Menezes et al. [B4])
LBA	logical block address
GCM	Galois/Counter Mode of authenticated encryption

4. Mathematical Conventions

This standard uses decimal, binary, and hexadecimal numbers. For clarity, decimal numbers generally represent counts, and binary or hexadecimal numbers describe bit patterns or raw binary data.

Binary numbers are represented by a string of one or more binary digits, followed by the subscript 2. For example, the decimal number 26 is represented as 00011010_2 in binary.

Hexadecimal numbers are represented by a string of one or more hexadecimal digits, prefixed by the string "0x". For example, the decimal number 135 is represented as 0x87 in hexadecimal.

~~The main procedures used in this standard are the AES block cipher encryption, the AES block cipher decryption, the multiplication-by-alpha, and the multiplication over the field GF (2^{128}).~~

The AES block cipher encryption of the value X with the key K is denoted as AES-enc(K, X), and the AES block cipher decryption is denoted as AES-Dec(K, X).

The multiplication-by-alpha of a 16-byte value $X \in$ GF (2^{128}) by a primitive element α in the field GF(2^{128}) is denoted as mult-by-alpha(X) and is defined in Section 5.2.1.

The multiplication of two elements $X, Y \in$ GF (2^{128}) is denoted as $X \cdot Y$, and the addition of X and Y is denoted as $X \oplus Y$. Addition in GF (2^{128}) is equivalent to the bitwise exclusive-or operation, and the multiplication operation is defined in Section 5.3.3. We denote the number of bits in a bit string X as #X.

The procedure length(S) returns a 64-bit string containing the nonnegative integer describing the number of bits in its argument S, with the most significant bit on the left and the least significant bit on the right. The expression 0^n denotes a string of n zero bits, and A|B denotes the concatenation of two bit strings A and B. The procedure $\text{msb}_t(S)$ returns the initial t bits of the string S. Bit strings are indexed starting on the left, so that bit zero of S is the leftmost bit. S[a:b] denotes the substring of S from the a^{th} bit through the b^{th}.

5. Wide-block Encryption Algorithms

5.1 Data Units and Associated Data Units

The purpose of this standard is to specify length-preserving encryption with associated data (EAD) algorithms that are suitable for the encryption of data at rest. An EAD algorithm consists of an encryption procedure and a decryption procedure. The encryption procedure accepts three inputs: a secret key, a plaintext, and the associated data. It returns a single ciphertext value. Each of these inputs is regarded as an octet string.

The secret key input must be unpredictable to the adversary. Each EAD algorithm accepts a key of a fixed length, but different algorithms may have keys of different lengths.

The plaintext input contains the data to be encrypted. Plaintext is divided into data units that, within a particular key scope, may have different lengths. An EAD algorithm defines the range of admissible plaintext lengths.

The associated data input contains data that is associated with the plaintext, but which does not need to be encrypted. The choice of data for this input is described in more detail below. Within a particular key scope, Associated data units may have different lengths.

The ciphertext returned by the encryption procedure is the same length as the plaintext.

The decryption procedure accepts the same three input described above: a secret key, a ciphertext, and the associated data value. It returns a single plaintext value.

The decryption procedure is the reverse of the encryption procedure; more specifically, if the encryption of the plaintext P with the key K and the associated data A results in the ciphertext C, then the decryption of C with the key K and the associated data A will result in the plaintext P.

This value of associated data must be known at the time of encryption and the time of decryption, so it should contain only information that is available, in plaintext form, at the time of both operations.

The associated data input should characterize the plaintext, and it should be as fine-grained as possible. This is because whenever the same plaintext is encrypted two different times using the same key but with distinct associated data values, the result is two distinct ciphertext values. Thus the use of distinct associated data values hides the equality of the plaintexts from an attacker.

5.1.1 Using EAD to protect a string of data block

An EAD can be used to protect a string of data blocks, such as those in a data-storage disk. In this application, the associated data input to the encryption and decryption procedure should contain the logical index of the block on which the procedure is acting. When this information is included in the associated data, cases in which two distinct data blocks contain identical plaintext values will be hidden from an adversary.

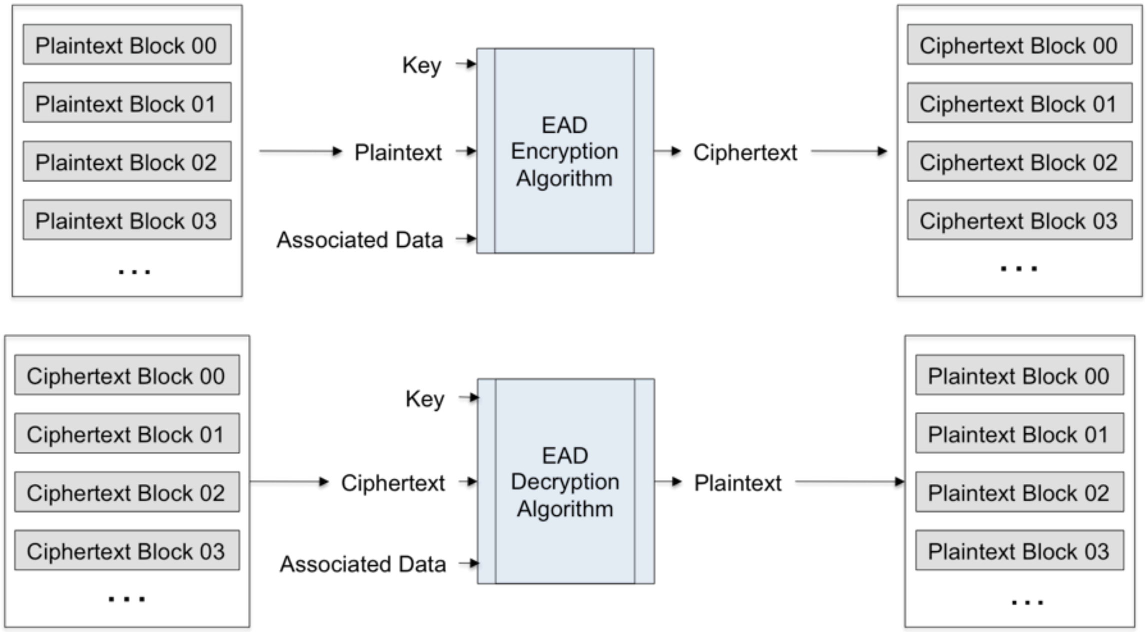

Figure 1-EAD Encryption and Decryption ~~of a Block Device~~

If multiple disks are being protected with a single secret key, then the associated data input should contain both the logical index of the block and an additional distinguishing parameter that is unique to each of the disks. When this information is included in the associated data, cases in which two distinct data blocks on different disks contain identical plaintext values will be hidden from an adversary.

5.2 The EME2-AES Transform

5.2.1 The mult-by-alpha Procedure

The encryption and decryption algorithms described in the following sections use a Mult-by-alpha(X) procedure that multiplies a 16-byte value X by a primitive element α in the field GF(2^{128}). The input value is first converted into a byte string X[i], i = 0,1,...,15, where X[0] is the first byte of the byte string.

The multiplication by alpha is defined by the following, or a mathematically equivalent, procedure:

```
Mult-by-alpha(X)
  Input:  byte string X[i], i = 0,1,...,15

  Output: byte string Y[i], i = 0,1,...,15

  for i=0 to 15 do
    Y[i] = 2*X[i] mod 256
    if (i>0 and X[i-1]>127) then Y[i]=Y[i]+1
  end-for
  if (X[15] > 127) then Y[0] = Y[0] xor 0x87
```

Conceptually, the procedure is a left shift of each byte by one bit with carry propagating from one byte to the next. Also, if the 15th (last) byte shift results in a carry, a special value (~~hexadecimal~~ 0x87) is xor'ed into the first byte. This value is derived from the modulus of the Galois Field (polynomial $x^{128}+x^7+x^2+x+1$).

5.2.2 EME2-AES Encryption

The EME2-AES encryption procedure can be described by the formula:

$$C = \text{EME2-AES-Enc}(Key, T, P)$$

where

Key is the 48 or 64 byte EME2-AES key

T is the value of the associated data, of arbitrary byte length (zero or more bytes)

P is the plaintext, of length 16 bytes or more

C is the ciphertext resulting from the operation, of the same byte-length as *P*

The input to the EME2-AES encryption procedure is parsed as follows:

- The key is partitioned into three fields, Key = $K_{AD} \mid K_{ECB} \mid K_{AES}$, with K_{AD} (the associated data key) consisting of the first 16 bytes, K_{ECB} (the ECB pass key) consisting of the following 16 bytes, and K_{AES} (the AES encrrytion/decryption key) consisting of the remaining 16 or 32 bytes.
- If not empty, the associated data T is partitioned into a sequence of blocks $T = T_1 \mid T_2 \mid \ldots \mid T_r$, where each of the blocks $T_1, T_2, \ldots, T_{r-1}$ is of length exactly 16 bytes, and T_r is of length between 1 and 16 bytes.
- The plaintext P is partitioned into a sequence of blocks $P = P_1 \mid P_2 \mid \ldots \mid P_m$, where each of the blocks $P_1, P_2, \ldots, P_{m-1}$ is of length exactly 16 bytes, and P_m is of length between 1 and 16 bytes.

The ciphertext shall then be computed by the sequence of steps in and or equivalent. An illustration of these steps (for plaintext of 130 full blocks and one partial block) is provided in Figure 2.

Table 1-The function H for processing the associated data T

```
// Function H($K_{AES}$, $K_{AD}$, T = ($T_1$ ... $T_{r-1}$ $T_r$) ) :
if len(T) == 0 then
    T_star = AES-Enc($K_{AES}$, $K_{AD}$)
else
  $K_{AD}$ = Mult-by-alpha($K_{AD}$)
    for j = 1 to r-1
        $TT_j$ = AES-Enc($K_{AES}$, $K_{AD} \oplus T_j$) $\oplus K_{AD}$
        $K_{AD}$ = Mult-by-alpha($K_{AD}$)
    if len($T_r$) < 16 then
        $T_r$ = $T_r$ | 0x80 0x00...     // pad $T_r$ to 16 bytes, 0x80 followed by 0's
        $K_{AD}$ = Mult-by-alpha($K_{AD}$)
    $TT_r$ = AES-Enc($K_{AES}$, $K_{AD} \oplus T_r$) $\oplus K_{AD}$
```

```
T_star = $TT_1 \oplus TT_2 \oplus \ldots \oplus TT_r$

return T_star                        // return the 16 byte value T_star
```

Table 2-The EME2-AES Encryption Procedure

```
// Function EME2-AES-Enc($K_{AES}$, $K_{ECB}$, $K_{AD}$, T, P = ($P_1$ ... $P_{m-1}$ $P_m$) ):
T_star = H($K_{AES}$, $K_{AD}$, T)        // Process the associated data
if len($P_m$) = 16 then
  lastFull = m
else
  lastFull = m-1
  $PPP_m$ = $P_m$ | 0x80 0x00 ...     // Pad $P_m$ to 16 bytes, 0x80 followed by 0's

//    First ECB pass
L = $K_{ECB}$
for j = 1 to lastFull
  $PPP_j$ = AES-Enc($K_{AES}$, L $\oplus$ $P_j$)
  L = Mult-by-alpha(L)

//    Intermediate mixing
MP = $PPP_1 \oplus PPP_2 \oplus \ldots \oplus PPP_m \oplus$ T_star
if len($P_m$) < 16 then
  MM = AES-Enc($K_{AES}$, MP)
  MC = $MC_1$ = AES-Enc($K_{AES}$, MM)
else
  MC = $MC_1$ = AES-Enc($K_{AES}$, MP)
M = $M_1$ = MP $\oplus$ MC
for j = 2 to lastFull
  if (j-1 mod 128 > 0) then       // use the current mask M
    M = Mult-by-alpha(M)
    $CCC_j$ = $PPP_j \oplus$ M
  else                            // calculate a new mask M
    MP = $PPP_i \oplus M_1$
    MC = AES-Enc($K_{AES}$, MP)
    M = MP $\oplus$ MC
    $CCC_j$ = MC $\oplus M_1$
if lastFull < m then
  $C_m$ = $P_m$ $\oplus$ (MM truncated to len($P_m$) bytes)
  $CCC_m$ = $C_m$ | 0x80 0x00 ...     // Pad $C_m$ to 16 bytes, 0x80 followed by 0's
$CCC_1$ = $MC_1 \oplus CCC_2 \oplus \ldots \oplus CCC_m \oplus$ T_star


//    Second ECB Pass
L = $K_{ECB}$
for j = 1 to lastFull
  $C_j$ = AES-Enc($K_{AES}$, $CCC_j$) $\oplus$ L
  L = Mult-by-alpha(L)

return C = ($C_1$ ... $C_{m-1}$ $C_m$)
```

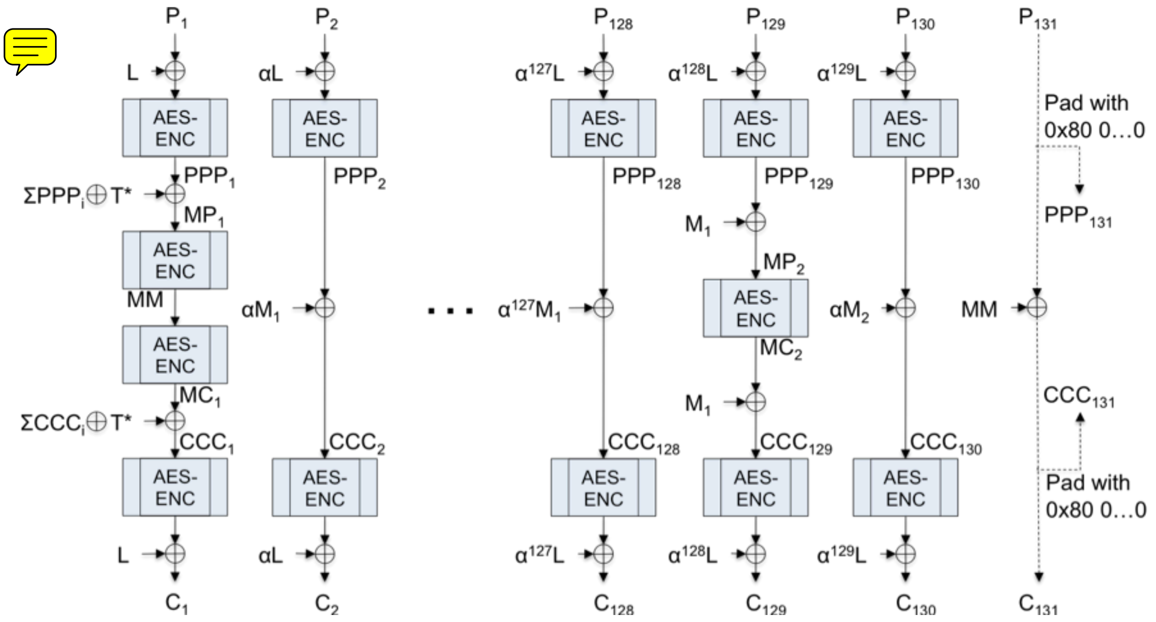

Figure 3-An illustration of EME2-AES encryption: each AES-ENC block has a Key input that is not shown. T* is computed from the associated data using the function H from , and the M_i's are computed as $M_i = MP_i$ xor MC_i

5.2.3 EME2-AES Decryption

The EME2-AES decryption procedure can be described by the formula:

$$P = \text{EME2-AES-Dec}(Key, T, C),$$

where

Key is the 48 or 64 byte EME2-AES key

T is the value of the associated data, of arbitrary byte length (zero or more bytes)

C is the ciphertext, of length 16 bytes or more

P is the plaintext resulting from the operation, of the same byte-length as *C*

The input to the EME2-AES decryption procedure is parsed as follows:

- The key is partitioned into three fields, Key = $K_{AD} \mid K_{ECB} \mid K_{AES}$, with K_{AD} (the associated data key) consisting of the first 16 bytes, K_{ECB} (the ECB pass key) consisting of the following 16 bytes, and K_{AES} (the AES encryption/decryption key) consisting of the remaining 16 or 32 bytes.

- If not empty, the associated data is partitioned into a sequence of blocks $T = T_1 | T_2 | \ldots | T_r$, where each of the blocks $T_1, T_2, \ldots, T_{r-1}$ is of length exactly 16 bytes, and T_r is of length between 1 and 16 bytes.
- The ciphertext P is partitioned into a sequence of blocks $C = C_1 | C_2 | \ldots | C_m$, where each of the blocks $C_1, C_2, \ldots, C_{m-1}$ is of length exactly 16 bytes, and C_m is of length between 1 and 16 bytes.

The plaintext shall then be computed by the sequence of steps in Table 3 or equivalent.

NOTE 1—The only difference between the encryption and decryption procedures is that all the AES-Enc calls in are replaced in Table 3 by calls to AES-Dec. The function H is defined in Table 1.

Table 3-The EME2-AES Decryption procedure

```
// Function EME2-AES-Dec($K_{AES}$, $K_{ECB}$ , $K_{AD}$, T, C = ($C_1$ ... $C_{m-1}$ $C_m$) ):

T_star = H($K_{AES}$, $K_{AD}$, T)        // Process the associated data
if len($C_m$) = 16 then
    lastFull = m
else
    lastFull = m-1
    $CCC_m$ = $C_m$ | 0x80 0x00 ...   // Pad $C_m$ to 16 bytes, 0x80 followed by 0's

// First ECB pass
L = $K_{ECB}$
for j = 1 to lastFull
    $CCC_j$ = AES-Enc($K_{AES}$, L $\oplus$ $C_j$)
    L = Mult-by-alpha(L)

// Intermediate mixing
MC = $CCC_1$ $\oplus$ $CCC_2$ $\oplus$ ... $\oplus$ $CCC_m$ $\oplus$ T_star
if len($C_m$) < 16 then
    MM = AES-Dec($K_{AES}$, MC)
    MP = $MP_1$ = AES-Dec($K_{AES}$, MM)
else
    MP = $MP_1$ = AES-Dec($K_{AES}$, MC)
M = $M_1$ = MP $\oplus$ MC
for j = 2 to lastFull
    if (j-1 mod 128 > 0) then      // use the current mask M
        M = Mult-by-alpha(M)
        $PPP_j$ = $CCC_j$ $\oplus$ M
    else                           // calculate a new mask M
        MC = $CCC_j$ $\oplus$ $M_1$
        MP = AES-Dec($K_{AES}$, MC)
        M = MP $\oplus$ MC
        $PPP_j$ = MP $\oplus$ $M_1$
if lastFull < m then
    $P_m$ = $C_m$ $\oplus$ (MM truncated to len($C_m$) bytes)
    $PPP_m$ = $P_m$ | 0x80 0x00 ...     // Pad $P_m$ to 16 bytes, 0x80 followed by 0's
$PPP_1$ = $MP_1$ $\oplus$ $PPP_2$ $\oplus$ ... $\oplus$ $PPP_m$ $\oplus$ T_star

// Second ECB Pass
L = $K_{ECB}$
for j = 1 to lastFull
    $P_j$ = AES-Dec($K_{AES}$, $PPP_j$) $\oplus$ L
    L = Mult-by-alpha(L)

return P = ($P_1$ ... $P_{m-1}$ $P_m$)
```

5.3 The XCB-AES Transform

5.3.1 Definition

The XCB-AES encryption and decryption algorithms use the AES block cipher encryption procedures AES-Enc and AES-Dec, as well as the hash function h and the pseudorandom function c. The variables H, K_e , K_d and K_c are derived from K, essentially by running the AES-Enc encryption procedure in CTR mode (see [N3]).

Optionally, these values can be stored between evaluations of these algorithms, in order to trade off some storage for a decreased computational load.

Let k be the size of the key fed to the AES procedure (either 16 or 32 bytes).

The function **c**: $\{0, 1\}^k \times \{0, 1\}^{128} \rightarrow \{0, 1\}^l$, where the output length l is bounded by $0 <= l <= 2^{39}$, generates an arbitrary-length output by running the AES-Enc procedure in counter mode, using its 16-byte input as the initial counter value. Its definition is

$$\mathbf{c}(K,W,l) = \text{AES-Enc}(K, W)|\text{AES-Enc}(K,\text{incr}(W))|. . .\text{msb}_t (\text{AES-Enc}(K,\text{incr}^{n-1}(W))),$$

where the output length l is indicated as an explicit parameter for clarity; $n = \lceil l/128 \rceil$ is the number of 16-byte blocks in the output and $t = l \bmod 128$ is number of bits in the trailing block.

Here the function incr : $\{0, 1\}^{128} \rightarrow \{0, 1\}^{128}$ is the increment operation that is used to generate successive counter values. This function treats the rightmost 32 bits of its argument as a nonnegative integer with the least significant bit on the right, increments this value modulo 2^{32}. More formally,

$$\text{incr}(X) = X[0; 95] \mid (X[96; 127] + 1 \bmod 2^{32}),$$

where bit strings are implicitly converted into integers.

The procedures h_1 and h_2 are defined in terms of the underlying hash function h as

$$h_1 (H, Z, B) = h(H, 0^{128}|Z,B|0^{\text{padlen1}(\#B)})$$

$$h_2 (H, Z, B) = h(H, Z |0^{128},B|0^{\text{padlen2}(\#B)} |\text{length}(Z|0^{128})|\text{length}(B))$$

The procedure padlen2(x) returns the smallest number that can be added to x so that the result is a multiple of 128. The procedure padlen1(x) returns padlen2(x) + 128. These procedures can be expressed mathematically as

$$\text{padlen2}(x) = 128 * \text{ceil}(x/128) - x$$

$$\text{padlen1}(x) = 128 * (1 + \text{ceil}(x/128)) – x = 128 + \text{padlen2}()$$

where ceil(x) is the smallest integer that is larger than x.

The procedure h : $\{0, 1\}^{128} \times \{0, 1\}^{a} \times \{0, 1\}^{c} \rightarrow \{0, 1\}^{128}$ is defined by $h(H, A, C) = X_{m+n+1}$, where a and c are within the interval $[128,2^{39}]$, and the variables $X_i \in \{0, 1\}^{128}$ for $i = 0,\ldots,m+n+1$ are defined as

$$
\begin{array}{ll}
X_i = 0 & \text{for } i = 0 \\
X_i = (X_{i-1} \oplus A_i) \cdot H & \text{for } i = 1, \ldots, m-1 \\
X_i = (X_{m-1} \oplus A_m) \cdot H & \text{for } i = m \\
X_i = (X_{i-1} \oplus C_{i-m}) \cdot H & \text{for } i = m+1, \ldots, m+n-1 \\
X_i = (X_{m+n-1} \oplus C_n) \cdot H & \text{for } i = m+n \\
X_i = (X_{m+n} \oplus (\text{length}(A) \mid \text{length}(C))) \cdot H & \text{for } i = m+n+1
\end{array}
$$

Here we let A_i denote the 16-byte substring A[128*(i-1); 128*i -1], and let C_i denote C [128*(i-1); 128*i-1]. In other words, A_i and C_i are the ith blocks of A and C, respectively, if those bit strings are decomposed into 16-byte blocks (the last A_i and C_i blocks will be padded with 0s, if shorter then 16 bytes).

NOTE 2—This procedure is identical to the universal hash function that is used as a component of the Galois/Counter Mode (GCM) of Operation [B3]. ~~(~~It is equivalent to the procedure used in Step 5 of Algorithm 4 of that specification, but please note that it is different than GHASH as defined in that document.~~)~~

5.3.2 Multiplication in GF(2^{128})

The multiplication operation of two 16-byte values X, Y $\in$ GF(2^{128}) is mathematically equivalent to an operation on bit vectors. The result Z = X $\cdot$ Y is also an element of GF(2^{128}). The input values are first converted into a byte string X[i], i = 0,1,...,15, where the leftmost bit is X[0] ~~,~~ and the rightmost bit is X[127] .

The multiplication in GF(2^{128}) operation is defined by the following, or a mathematically equivalent, procedure:

Table 4-The Multiplication in GF(2^{128}) Procedure

```
// compute Z = X * Y, where X, Y, Z are elements of GF(2^128) using an
// octet based algorithm
//
// here A[i] denotes the ith octet of A, and indexing starts at 0
//
// i and j are integers used in loops
// mask, b, and msb are unsigned integers

   /* initialize z to the all-zero element */
          for i = 0 to 15
             z[i] = 0

          for i=0 to 15   /* loop over bytes of y */
             mask = 128;
             while mask > 0       /* loop over bits in byte */
             /* if masked bit is set, add in terms from x */
                set b to y[i] & mask
                if b != 0
                   for j=0 to 15
                      set z[j] tp z[j] ^ x[j];
                mask = mask / 2;
             /* now execute LFSR shift on x  */
                set msb to x[15] & 0x01
                for j=15 down to 1
                   set b to x[j-1] & 0x01
                   set x[j] to (x[j] / 2) + b * 128
                set x[0] to x[0] / 2
                if msb = 1
                   set x[0] to x[0] ^ 0xe1
          return z
```

NOTE 3—The multiplication operation uses the special element R = $11100001_2|0^{120}$. The procedure rightshift() moves the bits of its argument one bit to the right. More formally, whenever W = rightshift(V), then W[i] = V[i-1] for $1 \le i \le 127$ and W[0] = 0.

5.3.3 XCB-AES Encryption

The XCB-AES encryption procedure for an m-bit block P is modeled with this equation:

$$CT \leftarrow \text{XCB-AES-Enc}(K,P,Z)$$

where:

K is either the 16 or 32 bytes XCB-AES key

P is a block of plaintext of m bits where $m \in [128, 2^{32}]$

Z is the value of the associated data, of arbitrary byte length (zero or more bytes)

CT is the block of 16 bytes of ciphertext resulting from the operation

The ciphertext shall then be computed by the following or an equivalent sequence of steps (see Figure 4):

$H \leftarrow \text{AES-Enc}(K, 0^{128})$

$K_e \leftarrow \text{msb}_k(\text{AES-Enc}(K, 0^{125}|001_2)|\ \text{AES-Enc}(K, 0^{125}|010_2))$

$K_d \leftarrow \text{msb}_k(\text{AES-Enc}(K, 0^{125}|011_2)|\ \text{AES-Enc}(K, 0^{125}|100_2))$

$K_c \leftarrow \text{msb}_k(\text{AES-Enc}(K, 0^{125}|101_2)|\ \text{AES-Enc}(K, 0^{125}|110_2))$

$A \leftarrow P[m-128; m-1]$

$B \leftarrow P[0; m-127]$

$C \leftarrow \text{AES-Enc}(K_e, A)$

$D \leftarrow C \oplus h_1(H, Z, B)$

$E \leftarrow B \oplus c(K_c, D, \#B)$

$F \leftarrow D \oplus h_2(H, Z, E)$

$G \leftarrow \text{AES-Dec}(K_d, F)$

$CT \leftarrow E|G$

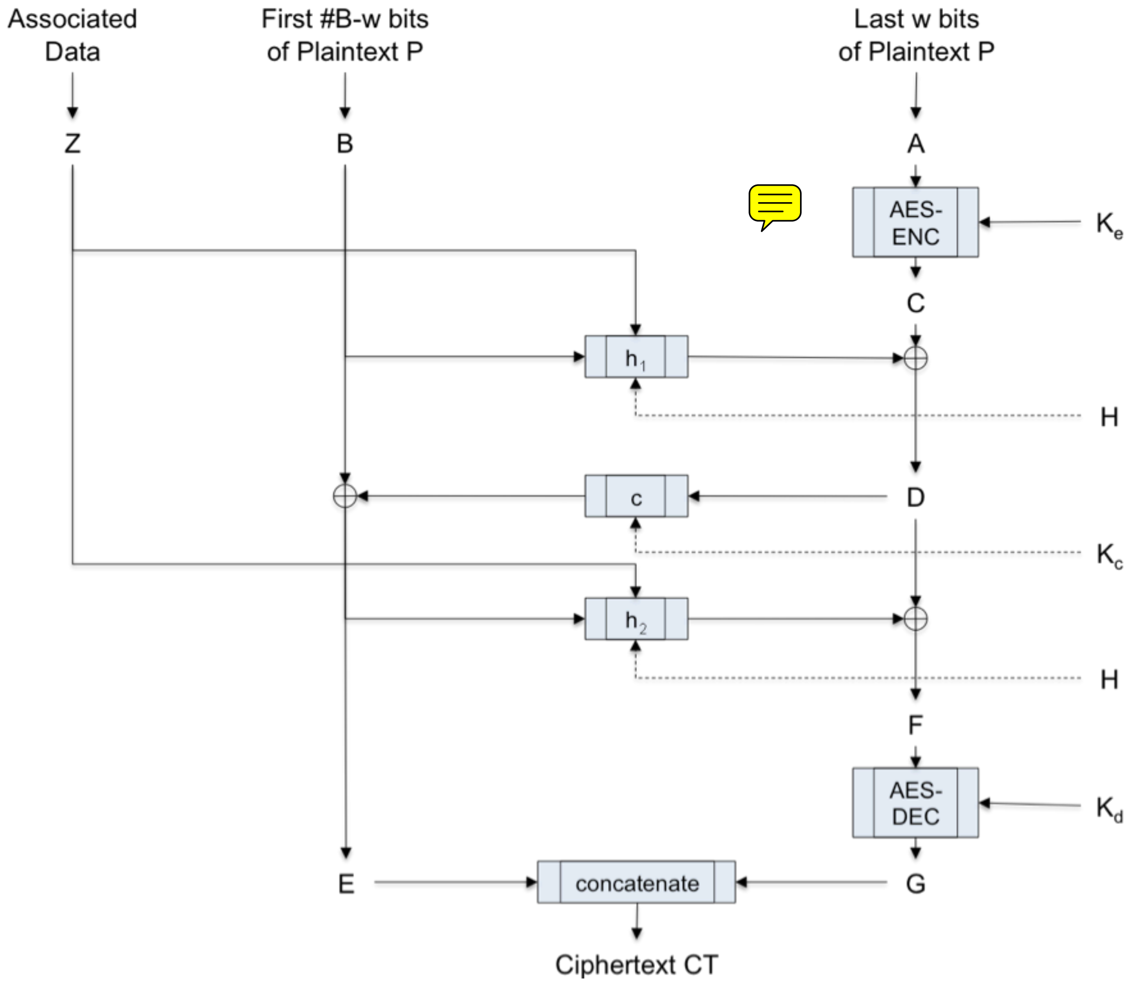

Figure 4-An illustration of XCB-AES encryption

5.3.4 XCB-AES Decryption

The XCB-AES decryption procedure for an m-bit block P is modeled with this equation:

$$P \leftarrow \text{XCB-AES-Dec}(K,CT,Z)$$

where:

K is either the 16 or 32 bytes XCB-AES key

Z is the value of the associated data, of arbitrary byte length (zero or more bytes)

CT is a block of ciphertext of m bits where $m \in [128,2^{32}]$

P is the block of 16 bytes of plaintext resulting from the operation

The plaintext shall then be computed by the following or an equivalent sequence of steps:

```
$H \leftarrow \text{AES-Enc}(K,0^{128})$
$K_e \leftarrow \text{msb}_k(\text{AES-Enc}(K,0^{125}|001_2)|\ \text{AES-Enc}(K,0^{125}|010))$
$K_d \leftarrow \text{msb}_k(\text{AES-Enc}(K,0^{125}|011_2)|\ \text{AES-Enc}(K,0^{125}|100_2))$
$K_c \leftarrow \text{msb}_k(\text{AES-Enc}(K,0^{125}|101_2)|\ \text{AES-Enc}(K,0^{125}|110_2))$
$G \leftarrow P[m-128; m-1]$
$E \leftarrow P[0; m-127]$
$F \leftarrow \text{AES-Enc}(K_d,A)$
$D \leftarrow F \oplus h_2(H,Z,E)$
$B \leftarrow E \oplus c(K_c,D,\#B)$
$C \leftarrow D \oplus h_1(H,Z,B)$
$A \leftarrow \text{AES-Dec}(K_e,F)$
$P \leftarrow B|A$
```

6. Use of Wide-Block Encryption for Storage

The encryption and decryption procedures described in 5.2.2 and 5.2.3 use AES as the basic building block with a key of either 48 or 64 bytes. The first mode shall be referred to as EME2-AES-384 and the second as EME2-AES-512.

The encryption and decryption procedures described in 5.3.2 and 5.3.4 use AES as the basic building block with a key of either 16 or 32 bytes. The first mode shall be referred to as XCB-AES-128 and the second as XCB-AES-256.

To be compliant with this standard, the implementation shall support at least one of the modes described in this standard.

In an application of this standard to sector-level encryption of a disk,

a) the data unit typically corresponds to a logical block,

b) the key scope typically includes a range of consecutive logical blocks on the disk, and

c) the associated data value corresponding to the first data unit in the scope typically corresponds to the Logical Block Address (LBA) associated with the logical block in the range.

The associated data values are assigned consecutively, starting from an arbitrary non-negative integer. When encrypting an associated data value using AES, the associated data value is first converted into a little-endian byte string. For example the associated data value 0x123456789a corresponds to byte string 0x9a, 0x78, 0x56, 0x34, 0x12.

A key used for wide-block encryption of storage shall not be associated with more than one key scope.

NOTE 4—The reason of the above restriction is that encrypting more than one block with the same key and the same associated data value introduces security vulnerabilities that might potentially be used in an attack on the system. In particular, key reuse enables trivial cut-and-paste attacks.

6.1 Selecting an EAD Algorithm

This document specifies two different EAD algorithms: EME2-AES and XCB-AES. Both modes implement a tweakable pseudorandom permutation with substantially similar security properties and have similar bounds with respect to the amount of data that can safely be encrypted with a single key.

Nevertheless, upon choosing a mode, implementers might need to consider other factors than security level: software performance or hardware implementation size are likely to be determinant factors.

In order to guide them in their choice of EAD, Table 5 shows a list of potentially significant differences, in term of computational complexity, between the two proposed algorithms, when encrypting data blocks made of n 16-byte cipher blocks:

Table 5-Differences, in term of computational complexity, between EME2-AES and XCB-AES

	EME2-AES	**XCB-AES**
Applications of the AES promitive	2n+1	n+1
shift and xor operations (multiplication by a fixed alpha)	3n	-
GF(2^128) multiplications	-	2n

Annex A

(informative)

Bibliography

[B1] Halevi, S., “EME*: extending EME to handle arbitrary-length messages with associated data,” INDOCRYPT 2004, Lecture Notes in Computer Science, vol. 3348, pp 315–327. Springer-Verlag, 2004.

[B2] McGrew, D., Fluhrer, S., “The Security of the Extended Codebook (XCB) Mode of Operation,” Proceedings of the 14th Annual Workshop on Selected Areas in Cryptography, Springer-Verlag, 2007.

[B3] NIST Draft Special Publication 800-38D (June 27, 2007), Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.

[B4] Menezes, A.J., Van Oorschot, P.C., Vanstone, S.A., Handbook of Applied Cryptography, CRC Press, October 1996.

Annex B

(informative)

Security Considerations

The security goal of length-preserving EAD can roughly be described as follows: First, fixing the plaintext length and any particular value for the associated data, the EAD scheme should look just like a block cipher with block size equals to the plaintext size. Namely, interacting with the encryption and decryption routines, it should be infeasible to distinguish them from a random permutation and its inverse. Moreover, varying the plaintext length and/or the associated data should look like using a different and independent key. (The term "wide block cipher" refers to the fact that the plaintext size can be larger than those typical of a block cipher.)

A little more precisely, the security model stipulates an attacker that can request the encryption of plaintext/associated-data pairs under an unknown key, and can similarly request the decryption of ciphertext/associated-data pairs under the same unknown key. These plaintext, ciphertext, and associated-data values can be adaptively chosen by the adversary. The security requirement asserts that this attacker cannot distinguish between the following two cases:

a) the queries are indeed answered by the scheme at hand with a fixed secret key.

b) the queries are answered by a set of random permutations, with different independent permutations for different values of the associated data and plaintext length.

This security requirement implies, in particular, that encrypting some plaintext with one value of the associated data and then decrypting the resulting ciphertext using a different value of the associated data must yield a (pseudo)random decrypted plaintext (since this is what happens in Case b).

Both EME2-AES and XCB-AES have been shown to be secure in this model, under the assumption that AES cannot be distinguished from a random permutation (and subject to some bound on the number of invocations with the same key - roughly the birthday bound).

This security model assumes that the encryption/decryption routines are never applied to plaintext/ciphertext values that relate directly to the secret keys. In particular this model does not consider the case where the encryption routine of the scheme is used to encrypt its own secret key. Neither EME2-AES nor XCB-AES has known vulnerabilities with respect to self-encryption of the secret keys, but to our knowledge neither of them was ever analyzed or proven secure in this model.

Annex C

(informative)

Implementation in C

Reference implementations of EME2 and XCB are available at the following website:

https://siswg.net/index.php?option=com_content&task=view&id=36&Itemid=75

Annex D

(informative)

Test Vectors

D.1 EME2-128 Test Case

[TBD]

D.2 XCB-AES Test Cases

The security analysis of XCB that was published at Selected Areas in Cryptography 2007 [B2] makes use of a fact that relates the internal variables F and C [1, Theorem 3]. This relation can be used as a consistency check on an implementation, since the relation must hold for each invocation of the XCB algorithm. Because the proof of security makes use of the fact that the relation holds, such a check connects the security analysis to the validation of the implementation, and thus provides added confidence of correctness. The test cases in this document have been verified against this consistency check.

Test vectors are encoded in C-array format in order to facilitate their use in C-code implementations. Each test vector is represented between brackets, and the following number is the number of octets. The C struct in use is shown in Table 6.

Table 6 - C struct used to represent test vectors

```
typedef struct test_case_t {
  uint8_t xcb_key[32];    /* can hold AES keys of any size (16, 24, 32) */
  uint8_t octets_in_key;
  uint8_t plaintext[TEST_BUF_LEN];
  unsigned int plaintext_len;
  uint8_t assoc_data[TEST_BUF_LEN];
  unsigned int assoc_data_len;
  uint8_t ciphertext[TEST_BUF_LEN];
  struct test_case_t *next;
} test_case_t;
```

Table 7 shows 8 test vectors referred as “case 0” through “case 7”. Case 1 tests XCB-AES-256, while the others test XCB-AES-128 with different plaintext lengths.

Table 7 - XCB-AES Test Cases

```
test_case_t case7 = {
  /* key */
  {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
  },
  /* octets in key */
  16,
```

```
  /* plaintext */
  {
    0x08, 0x47, 0x1e, 0x46, 0x29, 0x45, 0xa7, 0x41,
    0x54, 0x0f, 0xaa, 0x16, 0xf0, 0x1e, 0x42, 0x1b,
    0x7f, 0xa4, 0x3e, 0x0d, 0x1f, 0x99, 0xf6, 0xa0,
    0x1f, 0x71, 0x26, 0xf9, 0x8a, 0x3f, 0xc9, 0x6a,
    0xd6, 0x8b, 0xf8, 0x6e, 0xa8, 0xd7, 0x2a, 0xab,
    0x5d, 0x98, 0x7d, 0x08, 0x54, 0xea, 0x72, 0xfe,
    0xa7, 0x64, 0x3c, 0x65, 0x84, 0x33, 0xdd, 0x5e,
    0x31, 0xb4, 0x06, 0x70, 0xc6, 0xd6, 0x9d, 0x1b,
    0x4c, 0xe3, 0xac, 0x9d, 0x9f, 0x5f, 0x73, 0xc6,
    0x91, 0x8a, 0xeb, 0x8d, 0x4c, 0x2d, 0xad, 0xbe,
    0x12, 0xe6, 0xd0, 0xc7, 0x2f, 0x4c, 0xa9, 0x1e,
    0x66, 0xc6, 0xbe, 0xbd, 0x32, 0xf0, 0x09, 0x48,
    0x65, 0x81, 0xda, 0x90, 0x18, 0xa7, 0x4b, 0x9c,
    0x7e, 0x28, 0x8f, 0xb1, 0x8f, 0xd6, 0x09, 0x00,
    0xa4, 0x44, 0x8f, 0xab, 0xea, 0xd7, 0x3d, 0x13,
    0xcb, 0x24, 0x83, 0xfb, 0xc8, 0xfb, 0xdf, 0xe9,
    0x30, 0xa1, 0x38, 0x90, 0x55, 0x5c, 0xaa, 0x88,
    0xf4, 0xac, 0xdd, 0x5a, 0x3e, 0x51, 0x59, 0xe5,
    0xa6, 0x46, 0x7e, 0xc7, 0xef, 0x05, 0x23, 0x95,
    0x30, 0x14, 0xe6, 0xde, 0x79, 0x6c, 0xce, 0x7d,
    0x4f, 0xcd, 0x14, 0xb0, 0x67, 0x7a, 0x2d, 0x8e,
    0x50, 0x9f, 0x55, 0xc8, 0x14, 0xed, 0x12, 0xcd,
    0x75, 0x5c, 0xd8, 0xac, 0xb7, 0xbb, 0x12, 0x66,
    0xb4, 0xd7, 0x25, 0xe2, 0x50, 0x55, 0xe4, 0xd3,
    0x60, 0xb7, 0xcd, 0x31, 0xab, 0xdd, 0x5f, 0x42,
    0x92, 0x7a, 0x4c, 0x11, 0x16, 0x30, 0x5f, 0xea,
    0x7e, 0xcb, 0xac, 0x5d, 0xc4, 0x7f, 0xf2, 0xf3,
    0x30, 0xef, 0x10, 0x8d, 0xc8, 0x93, 0xf7, 0xbe,
    0xcd, 0x6e, 0xea, 0xa3, 0x95, 0x74, 0xdb, 0x1e,
    0xe8, 0x42, 0xea, 0xab, 0x10, 0xf1, 0x7c, 0x29,
    0x93, 0x1f, 0x92, 0x52, 0xc1, 0x0c, 0x40, 0x2c,
    0xaa, 0x00, 0xe8, 0x77, 0x2d, 0x54, 0x11, 0x1a,
    0xba, 0x50, 0x6e, 0x4f, 0xef, 0x24, 0x7b, 0x58,
    0xcb, 0x6a, 0xa2, 0xfc, 0xbb, 0xc4, 0xef, 0x91,
    0xc4, 0x04, 0x5d, 0xde, 0x51, 0x32, 0xda, 0x81,
    0x12, 0x12, 0x7c, 0xa4, 0xb0, 0x0b, 0x9c, 0xa9,
    0xa4, 0x28, 0x29, 0xa4, 0xd3, 0x9a, 0xaf, 0x2b,
    0xc1, 0x27, 0xd9, 0xe6, 0x9e, 0x92, 0x4f, 0x01,
    0x69, 0x29, 0xf9, 0x5f, 0x54, 0x68, 0xbe, 0x6f,
    0xc7, 0x41, 0x58, 0xe7, 0x0d, 0xa7, 0x9c, 0x74,
    0x83, 0x54, 0xab, 0x11, 0x81, 0xee, 0xbd, 0x77,
    0x47, 0xf8, 0xfb, 0x44, 0x08, 0x72, 0xd4, 0xb4,
    0xfb, 0xa2, 0x11, 0xfb, 0x4c, 0x00, 0x9a, 0xf0,
    0xd4, 0x1a, 0xc8, 0x13, 0x44, 0x11, 0x20, 0xb9,
    0x62, 0xde, 0x53, 0x01, 0xdd, 0x54, 0x4e, 0x0c,
    0x0b, 0x1a, 0xd4, 0x3f, 0x82, 0x9f, 0x76, 0xa5,
    0x1b, 0x33, 0x1c, 0xd4, 0x26, 0x51, 0xb6, 0xa2,
    0x26, 0x28, 0x42, 0xb9, 0x0c, 0xd2, 0x93, 0x24,
    0x18, 0xd8, 0xb6, 0x70, 0x75, 0x2a, 0x99, 0x25,
    0xd2, 0xfb, 0x80, 0xfa, 0x25, 0x23, 0xb4, 0x22,
    0x21, 0x21, 0xd0, 0x09, 0x99, 0x7e, 0xf2, 0x22,
    0x3a, 0xca, 0x4b, 0x12, 0xe6, 0x28, 0x05, 0x0d,
    0xce, 0x8d, 0x0a, 0x6b, 0xdc, 0xd5, 0x47, 0x49,
    0xe0, 0xda, 0x58, 0xf3, 0xfc, 0xa5, 0x63, 0x91,
    0xb5, 0x60, 0x2b, 0x5b, 0xbb, 0x13, 0xd0, 0xf1,
```

```
    0x2b, 0x1c, 0xd3, 0x0b, 0x45, 0xb6, 0xa7, 0x62,
    0x32, 0xdc, 0x27, 0xab, 0x81, 0x97, 0x1f, 0xab,
    0xdc, 0xc7, 0x5a, 0xee, 0x7b, 0xb6, 0x8b, 0xf9,
    0x35, 0x95, 0x55, 0xe2, 0x04, 0x8c, 0xd4, 0x4b,
    0x8e, 0x7a, 0xdb, 0x89, 0x52, 0xe2, 0xf0, 0xfa,
    0x3b, 0xda, 0x38, 0xbc, 0xa6, 0x49, 0x72, 0x4a,
    0x5f, 0x1d, 0x0a, 0xac, 0x41, 0x31, 0x0d, 0x75,
    0x78, 0xa6, 0x17, 0x48, 0x88, 0x82, 0xab, 0x66,
    0x3f, 0x46, 0x26, 0x19, 0x11, 0xe4, 0xb8, 0x41,
    0x27, 0xf3, 0x70, 0x62, 0x3b, 0x9f, 0xf6, 0x2e,
  },
  /* octets in plaintext */
  520,
  /* associated data */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in associated data */
  16,
  /* ciphertext */
  {
    0x28, 0xb0, 0xec, 0x43, 0x2f, 0x39, 0x7f, 0x1b,
    0x1a, 0xe9, 0x8e, 0x45, 0x86, 0xd2, 0x92, 0x66,
    0xae, 0x7e, 0x59, 0x78, 0x7c, 0x2d, 0x8e, 0x8b,
    0x3f, 0x3f, 0x1c, 0x10, 0xda, 0xfc, 0x7e, 0x63,
    0x13, 0x21, 0xec, 0x09, 0xe7, 0xa4, 0x7a, 0x04,
    0x92, 0xf1, 0xfb, 0x52, 0xff, 0x11, 0x23, 0xd4,
    0x96, 0xaf, 0xf0, 0xad, 0xbc, 0xb9, 0x32, 0x1c,
    0x9b, 0xd2, 0x91, 0x74, 0xc4, 0x78, 0x2b, 0x28,
    0xb1, 0x18, 0x92, 0x77, 0x72, 0x96, 0xd3, 0x0c,
    0xbc, 0xf0, 0x4f, 0x6e, 0x4f, 0x7a, 0xe6, 0x1a,
    0xc0, 0xa8, 0x6a, 0x06, 0x4c, 0xe9, 0xec, 0xe8,
    0x8b, 0x3a, 0x6d, 0x32, 0xd1, 0x79, 0xba, 0xca,
    0x91, 0x66, 0xcd, 0x15, 0xc5, 0xf1, 0x68, 0x7e,
    0x88, 0x9a, 0x1e, 0xe4, 0x0b, 0x32, 0x78, 0x3b,
    0x02, 0xdd, 0xfd, 0x50, 0x0b, 0x6c, 0xd4, 0x96,
    0xba, 0x1f, 0x5d, 0x7b, 0x6e, 0xd6, 0xfd, 0xee,
    0xfd, 0xc8, 0xc3, 0x6c, 0xa3, 0x81, 0x8b, 0x51,
    0x60, 0xb5, 0x58, 0x82, 0xc6, 0x16, 0x58, 0x03,
    0xdb, 0xbe, 0xe9, 0x5e, 0x12, 0xb5, 0xe2, 0xfd,
    0x4a, 0x0a, 0xfd, 0x5d, 0x84, 0x50, 0xd0, 0x98,
    0x3e, 0x30, 0xdb, 0x63, 0x18, 0x1f, 0x9a, 0x2a,
    0x3c, 0xc5, 0x16, 0xf2, 0x07, 0x59, 0x6e, 0xf5,
    0xee, 0x92, 0x7a, 0xfb, 0xf1, 0x41, 0xf0, 0xc5,
    0x5b, 0x0b, 0x08, 0x13, 0xe2, 0x99, 0x5b, 0x7c,
    0x4c, 0x13, 0xc0, 0x22, 0xe0, 0xba, 0x00, 0x42,
    0x27, 0x8b, 0x13, 0x32, 0x39, 0x1d, 0xb8, 0x9c,
    0x5d, 0xec, 0x68, 0x2f, 0xcd, 0xba, 0xdf, 0xba,
    0x6c, 0x01, 0x83, 0x25, 0x48, 0x47, 0x8f, 0x60,
    0x06, 0x21, 0x98, 0xa9, 0x5c, 0x85, 0xa3, 0xc8,
    0xf6, 0x33, 0x75, 0x3d, 0xc1, 0xe2, 0x9a, 0xc5,
    0x60, 0xf5, 0xf5, 0xf8, 0x1d, 0x9e, 0xaa, 0x24,
    0x00, 0x76, 0x65, 0x6b, 0x84, 0xe1, 0xd9, 0x20,
    0xb9, 0xd9, 0x68, 0xee, 0xb8, 0x4c, 0x74, 0x1a,
    0x22, 0x54, 0xe5, 0x11, 0x2c, 0x33, 0x92, 0xfb,
    0xd4, 0xf9, 0xb2, 0xdd, 0x30, 0x75, 0x2b, 0xf2,
```

```
    0x69, 0xef, 0x30, 0xa3, 0xca, 0x5c, 0x67, 0x35,
    0x6e, 0x4e, 0x53, 0xd9, 0xda, 0x6a, 0x1b, 0x99,
    0x55, 0x38, 0x1f, 0x85, 0x49, 0x1e, 0x52, 0xaa,
    0xdc, 0x38, 0xd8, 0x69, 0x61, 0xec, 0x53, 0x47,
    0xa7, 0x24, 0x04, 0xfc, 0x50, 0xd7, 0x33, 0x11,
    0xd8, 0x20, 0x00, 0x86, 0x98, 0x3e, 0x50, 0x35,
    0xff, 0x02, 0xb1, 0xf8, 0xf1, 0x44, 0xea, 0xef,
    0x31, 0x75, 0x12, 0x3a, 0xf4, 0x97, 0x0f, 0xc7,
    0x7e, 0x76, 0x91, 0xce, 0xe4, 0x50, 0x1d, 0x94,
    0x90, 0x69, 0xd6, 0x11, 0x6b, 0xf1, 0xb3, 0x01,
    0x2e, 0xac, 0x51, 0x07, 0x36, 0xc0, 0x9c, 0xfc,
    0x63, 0x6d, 0x01, 0x64, 0xf6, 0x9f, 0x52, 0x53,
    0xf4, 0xb4, 0x16, 0x2c, 0x5e, 0x55, 0x98, 0xcb,
    0x7b, 0x0f, 0x95, 0xff, 0xe4, 0xc0, 0x78, 0x97,
    0x1b, 0xe5, 0x49, 0x52, 0x0d, 0xec, 0x65, 0x5d,
    0xd6, 0x1d, 0x36, 0xcc, 0xa9, 0xd2, 0x6b, 0xaa,
    0x02, 0xb1, 0x8c, 0xed, 0x48, 0xfb, 0xee, 0xb4,
    0xb8, 0x42, 0xc0, 0x45, 0xc3, 0xc1, 0x18, 0x81,
    0xdc, 0x83, 0x76, 0xc5, 0xda, 0xfc, 0x82, 0xac,
    0xc6, 0xda, 0x45, 0x3a, 0xd3, 0xa1, 0x21, 0x39,
    0xab, 0x0f, 0x0f, 0x6d, 0xd7, 0xdf, 0x3b, 0x1e,
    0xe4, 0xaa, 0x71, 0x42, 0x8a, 0x19, 0xff, 0x97,
    0x31, 0x92, 0xeb, 0xd6, 0x0d, 0x6d, 0xe6, 0x98,
    0x84, 0xff, 0x99, 0xe9, 0x0d, 0xea, 0x4e, 0x5f,
    0xc0, 0xab, 0x0a, 0xa6, 0x0d, 0x96, 0x7d, 0x60,
    0x0b, 0xdd, 0x25, 0x9d, 0x5d, 0x63, 0xb3, 0xb9,
    0xd4, 0x85, 0x9e, 0xf7, 0x5d, 0x3d, 0xbd, 0xe2,
    0xd1, 0x4f, 0x17, 0x66, 0x07, 0xff, 0x3c, 0x1d,
    0xe5, 0xf6, 0x28, 0xc2, 0xfc, 0x65, 0x5f, 0x33,
    0x32, 0x29, 0xf7, 0x48, 0x12, 0x27, 0x98, 0xe3
  },
  NULL
};


test_case_t case6 = {
  /* key */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x16, 0x16, 0xdd, 0xa6
  },
  /* octets in key */
  16,
  /* plaintext */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
  },
  /* octets in plaintext */
  24,
  /* associated data */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in associated data */
```

```
  16,
  /* ciphertext */
  {
    0x70, 0x13, 0xfd, 0xe3, 0xc3, 0x9f, 0xa1, 0xa4,
    0x3f, 0x5a, 0xb4, 0x34, 0x5a, 0xbf, 0xe5, 0xd9,
    0xcf, 0x80, 0x85, 0xf8, 0x7e, 0xb3, 0x11, 0x89,
  },
  &case7
};

test_case_t case5 = {
  /* key */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x16, 0x16, 0xdd, 0xa6
  },
  /* octets in key */
  16,
  /* plaintext */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00
  },
  20,
  /* associated data */
  { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in associated data */
  16,
  /* ciphertext */
  {
    0x70, 0x13, 0xfd, 0xe3, 0xdb, 0x56, 0x19, 0xbf,
    0xa4, 0xed, 0x25, 0x6d, 0xb4, 0x44, 0x15, 0x68,
    0x7a, 0xa4, 0x50, 0x3f
  },
  &case6
};

test_case_t case4 = {
  /* key */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0xf3, 0x24, 0x6b, 0x19,
  },
  /* octets in key */
  16,
  /* plaintext */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in plaintext */
  16,
  /* associated data */
  {
```

```
    0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in associated data */
  16,
  /* ciphertext */
  {
    0x28, 0x2a, 0x71, 0x43, 0x39, 0xae, 0x66, 0x8c,
    0x3c, 0x20, 0x2a, 0xca, 0x9c, 0x71, 0xe0, 0x0b,
  },
  &case5
};

test_case_t case3 = {
  /* key */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x16, 0x16, 0xdd, 0xa6,
  },
  /* octets in key */
  16,
  /* plaintext */
  {
    0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /*  32 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /*  64 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /*  96 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 128 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 160 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 192 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 224 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 256 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 288 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 320 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 352 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 384 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 416 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 448 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /* 480 */
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00  /* 512 */
  },
  /* octets in plaintext */
  512,
  /* associated data */
  {
    0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in associated data */
  16,
  /* ciphertext */
  {
    0xbf, 0x2c, 0x04, 0x93, 0xbb, 0xb4, 0xbd, 0x55,
    0xcc, 0x11, 0xc0, 0x3d, 0xd9, 0x25, 0x1b, 0xe5,
    0x83, 0x79, 0x9f, 0x9d, 0xba, 0xcf, 0x23, 0x16,
    0x7a, 0x4c, 0x5e, 0xf0, 0x3e, 0x0d, 0xb9, 0x40,
    0x4e, 0x4e, 0xee, 0xb3, 0x5d, 0xdf, 0x15, 0x1d,
    0x23, 0x9e, 0x8b, 0x78, 0xc2, 0x64, 0x08, 0x24,
    0xce, 0x1f, 0x10, 0x6e, 0xab, 0x1c, 0x01, 0x9a,
    0xca, 0xd3, 0x98, 0x56, 0x31, 0xc7, 0x0c, 0x36,
    0x3f, 0x30, 0x15, 0xf5, 0xec, 0x41, 0xc8, 0x82,
    0x5e, 0xc4, 0xf4, 0x7f, 0x9e, 0xa0, 0x4d, 0x7e,
    0xdc, 0x17, 0x34, 0x1f, 0x5c, 0x41, 0x98, 0x9c,
    0x56, 0x3c, 0x6a, 0xc2, 0xac, 0x4e, 0xd8, 0xac,
    0x6b, 0xa4, 0x61, 0xfc, 0xaf, 0xb0, 0xb4, 0x1e,
    0x64, 0x4b, 0x00, 0x3c, 0xa3, 0xcf, 0x52, 0x60,
    0x73, 0xa1, 0xef, 0x97, 0x21, 0x7d, 0xf0, 0x3e,
```

```
    0x26, 0xbb, 0xd0, 0x22, 0xee, 0x27, 0x9f, 0x06,
    0x95, 0x3c, 0xa3, 0xcd, 0xfd, 0xb4, 0x3d, 0x49,
    0x20, 0xf3, 0x2e, 0xd6, 0x87, 0xd7, 0x81, 0x11,
    0x32, 0x84, 0xb1, 0x7d, 0x34, 0x10, 0x72, 0x58,
    0x1a, 0x3b, 0x38, 0xe7, 0x9f, 0x65, 0xd7, 0x54,
    0x9f, 0x80, 0x39, 0x00, 0x74, 0x5f, 0x37, 0x94,
    0xbf, 0x71, 0x75, 0xa8, 0xca, 0xeb, 0x62, 0xb7,
    0x96, 0x6f, 0xf7, 0xa2, 0xb7, 0x0f, 0xdf, 0x1f,
    0x12, 0x3f, 0x98, 0x26, 0x65, 0x2e, 0xda, 0x09,
    0x7e, 0x7f, 0x39, 0x2d, 0xf8, 0xd0, 0xa9, 0xc4,
    0xf4, 0x4b, 0xa4, 0x0e, 0x54, 0xb9, 0x71, 0xbe,
    0x31, 0x87, 0x6f, 0x1e, 0x43, 0xaa, 0x1f, 0x65,
    0xf5, 0xa6, 0x0e, 0xbf, 0x53, 0xf1, 0xea, 0x9b,
    0x8f, 0x9b, 0xc6, 0x37, 0x31, 0xfa, 0xbb, 0xb4,
    0xdf, 0xcb, 0xd2, 0xbc, 0xa9, 0x94, 0x70, 0x37,
    0x8f, 0x5a, 0x91, 0xc2, 0xf1, 0xbc, 0xb0, 0x80,
    0x10, 0xea, 0xfa, 0x3e, 0x32, 0xf3, 0xac, 0xe6,
    0xd3, 0xc9, 0xe9, 0x1d, 0x12, 0xd7, 0x9a, 0x78,
    0x3d, 0xb3, 0xf8, 0xdf, 0xec, 0xdd, 0xd8, 0x1a,
    0xda, 0xb8, 0x79, 0x03, 0x75, 0x28, 0x8c, 0x5d,
    0xf9, 0xee, 0xa4, 0xa6, 0x63, 0xb5, 0x45, 0x6a,
    0x02, 0xdc, 0x4f, 0xe4, 0x4c, 0xd9, 0x82, 0x1c,
    0x77, 0x3b, 0xdc, 0xfd, 0xf8, 0xc5, 0xe0, 0x68,
    0x65, 0x22, 0xab, 0x40, 0x98, 0x50, 0x01, 0x0f,
    0x34, 0xe9, 0x0a, 0x64, 0x2c, 0x0a, 0x96, 0xf2,
    0xbd, 0xa3, 0xe9, 0x75, 0x8b, 0xfd, 0xd5, 0x18,
    0x47, 0xa7, 0x15, 0xb0, 0xb8, 0xcf, 0x12, 0xc2,
    0x29, 0xf4, 0x39, 0x3d, 0xa6, 0xc8, 0x49, 0x72,
    0xf7, 0x3f, 0x2b, 0x2f, 0x72, 0xb7, 0x5d, 0x03,
    0x23, 0xe5, 0x9a, 0x48, 0xe3, 0xf2, 0x08, 0xe6,
    0x6d, 0xe7, 0x2f, 0x4d, 0x9a, 0x44, 0x04, 0x75,
    0x2a, 0xc7, 0x0f, 0x04, 0xe6, 0x47, 0x25, 0x27,
    0x1b, 0xd3, 0xff, 0xf2, 0x6c, 0xd7, 0xb4, 0x19,
    0x1d, 0x0d, 0xe3, 0xf7, 0x19, 0x63, 0xd7, 0x6e,
    0xf5, 0xda, 0x72, 0xbf, 0x7e, 0xf6, 0xd4, 0xdb,
    0xd7, 0x87, 0xce, 0xa1, 0x8a, 0x13, 0x6f, 0x01,
    0x2b, 0x2d, 0x8c, 0x8b, 0x50, 0x83, 0xdd, 0xcc,
    0xf8, 0xc2, 0x86, 0x41, 0xb6, 0x25, 0x60, 0x17,
    0x5f, 0x6d, 0x28, 0xea, 0xdd, 0xa5, 0xc9, 0xa1,
    0x5b, 0xf1, 0x53, 0xa5, 0xfd, 0x01, 0x16, 0xdf,
    0xd4, 0xf5, 0x62, 0x2a, 0x8f, 0x18, 0xd0, 0x7d,
    0x55, 0x93, 0x03, 0xe2, 0xe8, 0xdd, 0x10, 0x1c,
    0x17, 0x0f, 0xe8, 0x35, 0x88, 0xfb, 0xe2, 0x00,
    0x5e, 0x90, 0x07, 0x1b, 0xb0, 0x70, 0x64, 0xcd,
    0x36, 0x2e, 0x15, 0x32, 0x31, 0x1c, 0x06, 0x7e,
    0xf4, 0xa7, 0xa5, 0x00, 0xe3, 0x5e, 0x20, 0xc5,
    0x82, 0x05, 0x98, 0x18, 0xb3, 0x3e, 0xd0, 0x66,
    0x3f, 0x7a, 0xe0, 0xa0, 0xb2, 0xc8, 0x87, 0xef,
    0x72, 0x30, 0x91, 0x79, 0x9f, 0xaf, 0xfd, 0xbb,
  },
  &case4
};

test_case_t case2 = {
  /* key */
  {
```

```
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
  },
  /* octets in key */
  16,
  /* plaintext */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
  },
  /* octets in plaintext */
  48,
  /* associated data */
  { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in associated data */
  16,
  /* ciphertext */
  {
    0x97, 0xc6, 0xb2, 0xb7, 0x19, 0xa9, 0x54, 0xe3,
    0x3b, 0xab, 0x39, 0x0a, 0xf2, 0x57, 0xeb, 0x4c,
    0x59, 0x93, 0xdd, 0x9a, 0x1a, 0x36, 0x61, 0xd5,
    0xb1, 0x52, 0xf8, 0xd6, 0x5f, 0x35, 0x37, 0xb9,
    0x54, 0x34, 0xff, 0xf3, 0x35, 0x2d, 0xfe, 0xb6,
    0x61, 0x5e, 0xc1, 0xb1, 0xc6, 0x6d, 0x81, 0x5d,
  },
  &case3
};

test_case_t case1 = {
  /* key */
  {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
  },
  /* octets in key */
  32,
  /* plaintext */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

```
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
  },
  /* octets in plaintext */
  32,
  /* associated data */
  { 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in associated data */
  16,
  /* ciphertext */
  {
    0x0a, 0xa2, 0x7c, 0x16, 0x7b, 0x7a, 0x6f, 0x13,
    0x93, 0x23, 0x4c, 0xb1, 0x82, 0x8f, 0x73, 0x7c,
    0xe5, 0x3d, 0xa9, 0xf5, 0x05, 0x8e, 0xbd, 0x81,
    0xf4, 0x4b, 0xfb, 0x8a, 0xa6, 0x4a, 0xe6, 0xc1,
  },
  &case2
};

test_case_t case0 = {
  /* key */
  {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
  },
  /* octets in key */
  16,
  /* plaintext */
  {
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in plaintext */
  32,
  /* associated data */
  {
    0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  },
  /* octets in associated data */
  16,
  /* ciphertext */
  {
    0xf7, 0x27, 0xd7, 0x48, 0xb8, 0x6e, 0x3b, 0x36,
    0x2f, 0x20, 0x81, 0x0e, 0xed, 0xbe, 0x37, 0x8a,
    0x07, 0x76, 0x16, 0x31, 0xb9, 0x00, 0x94, 0x54,
    0xd5, 0x4d, 0x8d, 0x94, 0x9c, 0x35, 0x27, 0x19,
  },
  &case1
};
```