Internet Research Task Force (IRTF)
Request for Comments: 7253
Category: Informational
ISSN: 2070-1721

T. Krovetz
Sacramento State
P. Rogaway
UC Davis

May 2014

The OCB Authenticated-Encryption Algorithm

Abstract

This document specifies OCB, a shared-key blockcipher-based encryption scheme that provides confidentiality and authenticity for plaintexts and authenticity for associated data. This document is a product of the Crypto Forum Research Group (CFRG).

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Schemes for authenticated encryption (AE) simultaneously provide for confidentiality and authentication. While this goal would traditionally be achieved by melding separate encryption and authentication mechanisms, each using its own key, integrated AE schemes intertwine what is needed for confidentiality and what is needed for authenticity. By conceptualizing AE as a single cryptographic goal, AE schemes are less likely to be misused than conventional encryption schemes. Also, integrated AE schemes can be significantly faster than what one sees from composing separate confidentiality and authenticity means.

When an AE scheme allows for the authentication of unencrypted data at the same time that a plaintext is being encrypted and authenticated, the scheme is an authenticated encryption with associated data (AEAD) scheme. Associated data can be useful when, for example, a network packet has unencrypted routing information and an encrypted payload.

OCB is an AEAD scheme that depends on a blockcipher. This document fully defines OCB encryption and decryption except for the choice of the blockcipher and the length of authentication tag that is part of the ciphertext. The blockcipher must have a 128-bit blocksize. Each choice of blockcipher and tag length specifies a different variant of OCB. Several AES-based variants are defined in Section 3.1.

OCB encryption and decryption employ a nonce N, which must be distinct for each invocation of the OCB encryption operation. OCB requires the associated data A to be specified when one encrypts or decrypts, but it may be zero-length. The plaintext P and the associated data A can have any bitlength. The ciphertext C one gets by encrypting P in the presence of A consists of a ciphertext-core having the same length as P, plus an authentication tag. One can view the resulting ciphertext as either the pair (ciphertext-core, tag) or their concatenation (ciphertext-core || tag), the difference being purely how one assembles and parses ciphertexts. This document uses concatenation.

OCB encryption protects the confidentiality of P and the authenticity of A, N, and P. It does this using, on average, about a + m + 1.02 blockcipher calls, where a is the blocklength of A, m is the blocklength of P, and the nonce N is implemented as a counter (if N is random, then OCB uses a + m + 2 blockcipher calls). If A is fixed during a session, then, after preprocessing, there is effectively no cost to having A authenticated on subsequent encryptions, and the mode will average m + 1.02 blockcipher calls. OCB requires a single key K for the underlying blockcipher, and all blockcipher calls are keyed by K. OCB is online. In particular, one need not know the length of A or P to proceed with encryption, nor need one know the length of A or C to proceed with decryption. OCB is parallelizable: the bulk of its blockcipher calls can be performed simultaneously. Computational work beyond blockcipher calls consists of a small and fixed number of logical operations per call. OCB enjoys provable security: the mode of operation is secure assuming that the underlying blockcipher is secure. As with most modes of operation, security degrades as the number of blocks processed gets large (see Section 5 for details).

For reasons of generality, OCB is defined to operate on arbitrary bitstrings. But for reasons of simplicity and efficiency, most implementations will assume that strings operated on are bytestrings (i.e., strings that are a multiple of 8 bits). To promote interoperability, implementations of OCB that communicate with implementations of unknown capabilities should restrict all provided values (nonces, tags, plaintexts, ciphertexts, and associated data) to bytestrings.

The version of OCB defined in this document is a refinement of two prior schemes. The original OCB version was published in 2001 [OCB1] and was listed as an optional component in IEEE 802.11i. A second version was published in 2004 [OCB2] and is specified in ISO 19772. The scheme described here is called OCB3 in the 2011 paper describing the mode [OCB3]; it shall be referred to simply as OCB throughout this document. The only difference between the algorithm of this RFC and that of the [OCB3] paper is that the tag length is now encoded into the internally formatted nonce. See [OCB3] for complete references, timing information, and a discussion of the differences between the algorithms. OCB was initially the acronym for Offset Codebook but is now the algorithm's full name.

OCB has received years of in-depth analysis previous to its submission to the CFRG and has been under review by the members of the CFRG for over a year. It is the consensus of the CFRG that the security mechanisms provided by the OCB AEAD algorithm described in this document are suitable for use in providing confidentiality and authentication.

2. Notation and Basic Operations

There are two types of variables used in this specification, strings and integers. Although strings processed by most implementations of OCB will be strings of bytes, bit-level operations are used throughout this specification document for defining OCB. String variables are always written with an initial uppercase letter while integer variables are written in all lowercase. Following C's convention, a single equals ("=") indicates variable assignment and double equals ("==") is the equality relation. Whenever a variable is followed by an underscore ("_"), the underscore is intended to denote a subscript, with the subscripted expression requiring evaluation to resolve the meaning of the variable. For example, when i == 2, then P_i refers to the variable P_2.

c^i	The integer c raised to the i-th power.
bitlen(S)	The length of string S in bits (e.g., bitlen(101) == 3).
zeros(n)	The string made of n zero bits.
ntz(n)	The number of trailing zero bits in the base-2 representation of the positive integer n. More formally, ntz(n) is the largest integer x for which 2^x divides n.
S xor T	The string that is the bitwise exclusive-or of S and T. Strings S and T will always have the same length.
S[i]	The i-th bit of the string S (indices begin at 1, so if S is 011, then S[1] == 0, S[2] == 1, S[3] == 1).
S[i..j]	The substring of S consisting of bits i through j, inclusive.

S || T	String S concatenated with string T (e.g., 000 || 111 == 000111).
str2num(S)	The base-2 interpretation of bitstring S (e.g., str2num(1110) == 14).
num2str(i,n)	The n-bit string whose base-2 interpretation is i (e.g., num2str(14,4) == 1110 and num2str(1,2) == 01).
double(S)	If S[1] == 0, then double(S) == (S[2..128] || 0); otherwise, double(S) == (S[2..128] || 0) xor (zeros(120) || 10000111).

3. OCB Global Parameters

To be complete, the algorithms in this document require specification of two global parameters: a blockcipher operating on 128-bit blocks and the length of authentication tags in use.

Specifying a blockcipher implicitly defines the following symbols.

KEYLEN	The blockcipher’s key length in bits.
ENCIPHER(K,P)	The blockcipher function mapping 128-bit plaintext block P to its corresponding ciphertext block using KEYLEN-bit key K.
DECIPHER(K,C)	The inverse blockcipher function mapping 128-bit ciphertext block C to its corresponding plaintext block using KEYLEN-bit key K.

The TAGLEN parameter specifies the length of authentication tag used by OCB and may be any value up to 128. Greater values for TAGLEN provide greater assurances of authenticity, but ciphertexts produced by OCB are longer than their corresponding plaintext by TAGLEN bits. See Section 5 for details about TAGLEN and security.

As an example, if 128-bit authentication tags and AES with 192-bit keys are to be used, then KEYLEN is 192, ENCIPHER refers to the AES-192 cipher, DECIPHER refers to the AES-192 inverse cipher, and TAGLEN is 128 [AES].

3.1. Named OCB Parameter Sets and RFC 5116 Constants

The following table gives names to common OCB global parameter sets. Each of the AES variants is defined in [AES].

Name	Blockcipher	TAGLEN
AEAD_AES_128_OCB_TAGLEN128	AES-128	128
AEAD_AES_128_OCB_TAGLEN96	AES-128	96
AEAD_AES_128_OCB_TAGLEN64	AES-128	64
AEAD_AES_192_OCB_TAGLEN128	AES-192	128
AEAD_AES_192_OCB_TAGLEN96	AES-192	96
AEAD_AES_192_OCB_TAGLEN64	AES-192	64
AEAD_AES_256_OCB_TAGLEN128	AES-256	128
AEAD_AES_256_OCB_TAGLEN96	AES-256	96
AEAD_AES_256_OCB_TAGLEN64	AES-256	64

RFC 5116 defines an interface for authenticated-encryption schemes [RFC5116]. RFC 5116 requires the specification of certain constants for each named AEAD scheme. For each of the OCB parameter sets listed above: P_MAX, A_MAX, and C_MAX are all unbounded; N_MIN is 1 byte, and N_MAX is 15 bytes. The parameter sets indicating the use of AES-128, AES-192, and AES-256 have K_LEN equal to 16, 24, and 32 bytes, respectively.

Each ciphertext is longer than its corresponding plaintext by exactly TAGLEN bits, and TAGLEN is given at the end of each name. For instance, an AEAD_AES_128_OCB_TAGLEN64 ciphertext is exactly 64 bits longer than its corresponding plaintext.

4. OCB Algorithms

OCB is described in this section using pseudocode. Given any collection of inputs of the required types, following the pseudocode description for a function will produce the correct output of the promised type.

4.1. Processing Associated Data: HASH

OCB has the ability to authenticate unencrypted associated data at the same time that it provides for authentication and encrypts a plaintext. The following hash function is central to providing this functionality. If an application has no associated data, then the associated data should be considered to exist and to be the empty string. HASH, conveniently, always returns zeros(128) when the associated data is the empty string.

```
   Function name:
     HASH
   Input:
     K, string of KEYLEN bits                      // Key
     A, string of any length                       // Associated data
   Output:
     Sum, string of 128 bits                       // Hash result
```

Sum is defined as follows.

```
     //
     // Key-dependent variables
     //
     L_* = ENCIPHER(K, zeros(128))
     L_$ = double(L_*)
     L_0 = double(L_$)
     L_i = double(L_{i-1}) for every integer i > 0

     //
     // Consider A as a sequence of 128-bit blocks
     //
     Let m be the largest integer so that 128m <= bitlen(A)
     Let A_1, A_2, ..., A_m and A_* be strings so that
       A == A_1 || A_2 || ... || A_m || A_*, and
       bitlen(A_i) == 128 for each 1 <= i <= m.
       Note: A_* may possibly be the empty string.

     //
     // Process any whole blocks
     //
     Sum_0 = zeros(128)
     Offset_0 = zeros(128)
     for each 1 <= i <= m
        Offset_i = Offset_{i-1} xor L_{ntz(i)}
        Sum_i = Sum_{i-1} xor ENCIPHER(K, A_i xor Offset_i)
     end for

     //
     // Process any final partial block; compute final hash value
     //
     if bitlen(A_*) > 0 then
        Offset_* = Offset_m xor L_*
        CipherInput = (A_* || 1 || zeros(127-bitlen(A_*))) xor Offset_*
        Sum = Sum_m xor ENCIPHER(K, CipherInput)
     else
        Sum = Sum_m
     end if
```

4.2. Encryption: OCB-ENCRYPT

This function computes a ciphertext (which includes a bundled authentication tag) when given a plaintext, associated data, nonce, and key. For each invocation of OCB-ENCRYPT using the same key K, the value of the nonce input N must be distinct.

```
Function name:
  OCB-ENCRYPT
Input:
  K, string of KEYLEN bits                      // Key
  N, string of no more than 120 bits            // Nonce
  A, string of any length                       // Associated data
  P, string of any length                       // Plaintext
Output:
  C, string of length bitlen(P) + TAGLEN bits   // Ciphertext
```

C is defined as follows.

```
  //
  // Key-dependent variables
  //
  L_* = ENCIPHER(K, zeros(128))
  L_$ = double(L_*)
  L_0 = double(L_$)
  L_i = double(L_{i-1}) for every integer i > 0

  //
  // Consider P as a sequence of 128-bit blocks
  //
  Let m be the largest integer so that 128m <= bitlen(P)
  Let P_1, P_2, ..., P_m and P_* be strings so that
    P == P_1 || P_2 || ... || P_m || P_*, and
    bitlen(P_i) == 128 for each 1 <= i <= m.
    Note: P_* may possibly be the empty string.

  //
  // Nonce-dependent and per-encryption variables
  //
  Nonce = num2str(TAGLEN mod 128,7) || zeros(120-bitlen(N)) || 1 || N
  bottom = str2num(Nonce[123..128])
  Ktop = ENCIPHER(K, Nonce[1..122] || zeros(6))
  Stretch = Ktop || (Ktop[1..64] xor Ktop[9..72])
  Offset_0 = Stretch[1+bottom..128+bottom]
  Checksum_0 = zeros(128)
```

```
      //
      // Process any whole blocks
      //
      for each 1 <= i <= m
         Offset_i = Offset_{i-1} xor L_{ntz(i)}
         C_i = Offset_i xor ENCIPHER(K, P_i xor Offset_i)
         Checksum_i = Checksum_{i-1} xor P_i
      end for

      //
      // Process any final partial block and compute raw tag
      //
      if bitlen(P_*) > 0 then
         Offset_* = Offset_m xor L_*
         Pad = ENCIPHER(K, Offset_*)
         C_* = P_* xor Pad[1..bitlen(P_*)]
         Checksum_* = Checksum_m xor (P_* || 1 || zeros(127-bitlen(P_*)))
         Tag = ENCIPHER(K, Checksum_* xor Offset_* xor L_$) xor HASH(K,A)
      else
         C_* = <empty string>
         Tag = ENCIPHER(K, Checksum_m xor Offset_m xor L_$) xor HASH(K,A)
      end if

      //
      // Assemble ciphertext
      //
      C = C_1 || C_2 || ... || C_m || C_* || Tag[1..TAGLEN]
```

4.3. Decryption: OCB-DECRYPT

This function computes a plaintext when given a ciphertext, associated data, nonce, and key. An authentication tag is embedded in the ciphertext. If the tag is not correct for the ciphertext, associated data, nonce, and key, then an INVALID signal is produced.

```
   Function name:
     OCB-DECRYPT
   Input:
     K, string of KEYLEN bits                     // Key
     N, string of no more than 120 bits           // Nonce
     A, string of any length                      // Associated data
     C, string of at least TAGLEN bits            // Ciphertext
   Output:
     P, string of length bitlen(C) - TAGLEN bits,  // Plaintext
          or INVALID indicating authentication failure
```

P is defined as follows.

```
//
// Key-dependent variables
//
L_* = ENCIPHER(K, zeros(128))
L_$ = double(L_*)
L_0 = double(L_$)
L_i = double(L_{i-1}) for every integer i > 0

//
// Consider C as a sequence of 128-bit blocks
//
Let m be the largest integer so that 128m <= bitlen(C) - TAGLEN
Let C_1, C_2, ..., C_m, C_* and T be strings so that
  C == C_1 || C_2 || ... || C_m || C_* || T,
  bitlen(C_i) == 128 for each 1 <= i <= m, and
  bitlen(T) == TAGLEN.
  Note: C_* may possibly be the empty string.

//
// Nonce-dependent and per-decryption variables
//
Nonce = num2str(TAGLEN mod 128,7) || zeros(120-bitlen(N)) || 1 || N
bottom = str2num(Nonce[123..128])
Ktop = ENCIPHER(K, Nonce[1..122] || zeros(6))
Stretch = Ktop || (Ktop[1..64] xor Ktop[9..72])
Offset_0 = Stretch[1+bottom..128+bottom]
Checksum_0 = zeros(128)

//
// Process any whole blocks
//
for each 1 <= i <= m
   Offset_i = Offset_{i-1} xor L_{ntz(i)}
   P_i = Offset_i xor DECIPHER(K, C_i xor Offset_i)
   Checksum_i = Checksum_{i-1} xor P_i
end for

//
// Process any final partial block and compute raw tag
//
if bitlen(C_*) > 0 then
   Offset_* = Offset_m xor L_*
   Pad = ENCIPHER(K, Offset_*)
   P_* = C_* xor Pad[1..bitlen(C_*)]
   Checksum_* = Checksum_m xor (P_* || 1 || zeros(127-bitlen(P_*)))
   Tag = ENCIPHER(K, Checksum_* xor Offset_* xor L_$) xor HASH(K,A)
```

```
     else
        P_* = <empty string>
        Tag = ENCIPHER(K, Checksum_m xor Offset_m xor L_$) xor HASH(K,A)
     end if

     //
     // Check for validity and assemble plaintext
     //
     if (Tag[1..TAGLEN] == T) then
        P = P_1 || P_2 || ... || P_m || P_*
     else
        P = INVALID
     end if
```

5. Security Considerations

OCB achieves two security properties, confidentiality and authenticity. Confidentiality is defined via "indistinguishability from random bits", meaning that an adversary is unable to distinguish OCB outputs from an equal number of random bits. Authenticity is defined via "authenticity of ciphertexts", meaning that an adversary is unable to produce any valid nonce-ciphertext pair that it has not already acquired. The security guarantees depend on the underlying blockcipher being secure in the sense of a strong pseudorandom permutation. Thus, if OCB is used with a blockcipher that is not secure as a strong pseudorandom permutation, the security guarantees vanish. The need for the strong pseudorandom permutation property means that OCB should be used with a conservatively designed, well-trusted blockcipher, such as AES.

Both the confidentiality and the authenticity properties of OCB degrade as per s^2 / 2^128, where s is the total number of blocks that the adversary acquires. The consequence of this formula is that the proven security disappears when s becomes as large as 2^64. Thus, the user should never use a key to generate an amount of ciphertext that is near to, or exceeds, 2^64 blocks. In order to ensure that s^2 / 2^128 remains small, a given key should be used to encrypt at most 2^48 blocks (2^55 bits or 4 petabytes), including the associated data. To ensure these limits are not crossed, automated key management is recommended in systems exchanging large amounts of data [RFC4107].

When a ciphertext decrypts as INVALID, it is the implementor’s responsibility to make sure that no information beyond this fact is made adversarially available.

OCB encryption and decryption produce an internal 128-bit authentication tag. The parameter TAGLEN determines how many bits of this internal tag are included in ciphertexts and used for authentication. The value of TAGLEN has two impacts: an adversary can trivially forge with probability 2^{-TAGLEN}, and ciphertexts are TAGLEN bits longer than their corresponding plaintexts. It is up to the application designer to choose an appropriate value for TAGLEN. Long tags cost no more computationally than short ones.

Normally, a given key should be used to create ciphertexts with a single tag length, TAGLEN, and an application should reject any ciphertext that claims authenticity under the same key but a different tag length. While the ciphertext core and all of the bits of the tag do depend on the tag length, this is done for added robustness to misuse and should not suggest that receivers accept ciphertexts employing variable tag lengths under a single key.

Timing attacks are not a part of the formal security model and an implementation should take care to mitigate them in contexts where this is a concern. To render timing attacks impotent, the amount of time to encrypt or decrypt a string should be independent of the key and the contents of the string. The only explicitly conditional OCB operation that depends on private data is double(), which means that using constant-time blockcipher and double() implementations eliminates most (if not all) sources of timing attacks on OCB. Power-usage attacks are likewise out of the scope of the formal model and should be considered for environments where they are threatening.

The OCB encryption scheme reveals in the ciphertext the length of the plaintext. Sometimes the length of the plaintext is a valuable piece of information that should be hidden. For environments where "traffic analysis" is a concern, techniques beyond OCB encryption (typically involving padding) would be necessary.

Defining the ciphertext that results from OCB-ENCRYPT to be the pair (C_1 || C_2 || ... || C_m || C_*, Tag[1..TAGLEN]) instead of the concatenation C_1 || C_2 || ... || C_m || C_* || Tag[1..TAGLEN] introduces no security concerns. Because TAGLEN is fixed, both versions allow ciphertexts to be parsed unambiguously.

5.1. Nonce Requirements

It is crucial that, as one encrypts, one does not repeat a nonce. The inadvertent reuse of the same nonce by two invocations of the OCB encryption operation, with the same key, but with distinct plaintext values, undermines the confidentiality of the plaintexts protected in those two invocations and undermines all of the authenticity and integrity protection provided by that key. For this reason, OCB should only be used whenever nonce uniqueness can be provided with certainty. Note that it is acceptable to input the same nonce value multiple times to the decryption operation. We emphasize that the security consequences are quite serious if an attacker observes two ciphertexts that were created using the same nonce and key values, unless the plaintext and associated data values in both invocations of the encrypt operation were identical. First, a loss of confidentiality ensues because the attacker will be able to infer relationships between the two plaintext values. Second, a loss of authenticity ensues because the attacker will be able to recover secret information used to provide authenticity, making subsequent forgeries trivial. Note that there are AEAD schemes, particularly the Synthetic Initialization Vector (SIV) [RFC5297], appropriate for environments where nonces are unavailable or unreliable. OCB is not such a scheme.

Nonces need not be secret, and a counter may be used for them. If two parties send OCB-encrypted plaintexts to one another using the same key, then the space of nonces used by the two parties must be partitioned so that no nonce that could be used by one party to encrypt could be used by the other to encrypt (e.g., odd and even counters).

6. IANA Considerations

The Internet Assigned Numbers Authority (IANA) has defined a registry for Authenticated Encryption with Associated Data parameters. The IANA has added the following entries to the AEAD Registry. Each name refers to a set of parameters defined in Section 3.1.

Name	Reference	Numeric ID
AEAD_AES_128_OCB_TAGLEN128	Section 3.1	20
AEAD_AES_128_OCB_TAGLEN96	Section 3.1	21
AEAD_AES_128_OCB_TAGLEN64	Section 3.1	22
AEAD_AES_192_OCB_TAGLEN128	Section 3.1	23
AEAD_AES_192_OCB_TAGLEN96	Section 3.1	24
AEAD_AES_192_OCB_TAGLEN64	Section 3.1	25
AEAD_AES_256_OCB_TAGLEN128	Section 3.1	26
AEAD_AES_256_OCB_TAGLEN96	Section 3.1	27
AEAD_AES_256_OCB_TAGLEN64	Section 3.1	28

7. Acknowledgements

The design of the original OCB scheme [OCB1] was done while Rogaway was at Chiang Mai University, Thailand. Follow-up work [OCB2] was done with support of NSF grant 0208842 and a gift from Cisco. The final work by Krovetz and Rogaway [OCB3] that has resulted in this specification was supported by NSF grant 0904380. Thanks go to the many members of the Crypto Forum Research Group (CFRG) who provided feedback on earlier drafts. Thanks in particular go to David McGrew for contributing some text and for managing the RFC approval process, to James Manger for initiating a productive discussion on tag-length dependency and for greatly improving Appendix A, to Matt Caswell and Peter Dettman for writing implementations and verifying test vectors, and to Stephen Farrell and Spencer Dawkins for their careful reading and suggestions.

8. References

8.1. Normative References

[AES] National Institute of Standards and Technology, "Advanced Encryption Standard (AES)", FIPS PUB 197, November 2001.

[RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, January 2008.

8.2. Informative References

[OCB1] Rogaway, P., Bellare, M., Black, J., and T. Krovetz, "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption", ACM Conference on Computer and Communications Security 2001 - CCS 2001, ACM Press, 2001.

[OCB2] Rogaway, P., "Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC", Advances in Cryptology - ASIACRYPT 2004, Springer, 2004.

[OCB3] Krovetz, T. and P. Rogaway, "The Software Performance of Authenticated-Encryption Modes", Fast Software Encryption - FSE 2011 Springer, 2011.

[RFC4107] Bellovin, S. and R. Housley, "Guidelines for Cryptographic Key Management", BCP 107, RFC 4107, June 2005.

[RFC5297] Harkins, D., "Synthetic Initialization Vector (SIV) Authenticated Encryption Using the Advanced Encryption Standard (AES)", RFC 5297, October 2008.

Appendix A. Sample Results

This section gives sample output values for various inputs when using OCB with AES as per the parameters defined in Section 3.1. All strings are represented in hexadecimal (e.g., 0F represents the bitstring 00001111).

The following 16 (N,A,P,C) tuples show the ciphertext C that results from OCB-ENCRYPT(K,N,A,P) for various lengths of associated data (A) and plaintext (P). The key (K) has a fixed value, the tag length is 128 bits, and the nonce (N) increments.

```
  K : 000102030405060708090A0B0C0D0E0F
```

An empty entry indicates the empty string.

```
  N: BBAA99887766554433221100
  A:
  P:
  C: 785407BFFFC8AD9EDCC5520AC9111EE6

  N: BBAA99887766554433221101
  A: 0001020304050607
  P: 0001020304050607
  C: 6820B3657B6F615A5725BDA0D3B4EB3A257C9AF1F8F03009

  N: BBAA99887766554433221102
  A: 0001020304050607
  P:
  C: 81017F8203F081277152FADE694A0A00

  N: BBAA99887766554433221103
  A:
  P: 0001020304050607
  C: 45DD69F8F5AAE72414054CD1F35D82760B2CD00D2F99BFA9

  N: BBAA99887766554433221104
  A: 000102030405060708090A0B0C0D0E0F
  P: 000102030405060708090A0B0C0D0E0F
  C: 571D535B60B277188BE5147170A9A22C3AD7A4FF3835B8C5
     701C1CCEC8FC3358

  N: BBAA99887766554433221105
  A: 000102030405060708090A0B0C0D0E0F
  P:
  C: 8CF761B6902EF764462AD86498CA6B97
```

```
     N: BBAA99887766554433221106
     A:
     P: 000102030405060708090A0B0C0D0E0F
     C: 5CE88EC2E0692706A915C00AEB8B2396F40E1C743F52436B
        DF06D8FA1ECA343D

     N: BBAA99887766554433221107
     A: 000102030405060708090A0B0C0D0E0F1011121314151617
     P: 000102030405060708090A0B0C0D0E0F1011121314151617
     C: 1CA2207308C87C010756104D8840CE1952F09673A448A122
        C92C62241051F57356D7F3C90BB0E07F

     N: BBAA99887766554433221108
     A: 000102030405060708090A0B0C0D0E0F1011121314151617
     P:
     C: 6DC225A071FC1B9F7C69F93B0F1E10DE

     N: BBAA99887766554433221109
     A:
     P: 000102030405060708090A0B0C0D0E0F1011121314151617
     C: 221BD0DE7FA6FE993ECCD769460A0AF2D6CDED0C395B1C3C
        E725F32494B9F914D85C0B1EB38357FF

     N: BBAA9988776655443322110A
     A: 000102030405060708090A0B0C0D0E0F1011121314151617
        18191A1B1C1D1E1F
     P: 000102030405060708090A0B0C0D0E0F1011121314151617
        18191A1B1C1D1E1F
     C: BD6F6C496201C69296C11EFD138A467ABD3C707924B964DE
        AFFC40319AF5A48540FBBA186C5553C68AD9F592A79A4240

     N: BBAA9988776655443322110B
     A: 000102030405060708090A0B0C0D0E0F1011121314151617
        18191A1B1C1D1E1F
     P:
     C: FE80690BEE8A485D11F32965BC9D2A32

     N: BBAA9988776655443322110C
     A:
     P: 000102030405060708090A0B0C0D0E0F1011121314151617
        18191A1B1C1D1E1F
     C: 2942BFC773BDA23CABC6ACFD9BFD5835BD300F0973792EF4
        6040C53F1432BCDFB5E1DDE3BC18A5F840B52E653444D5DF
     N: BBAA9988776655443322110D
     A: 000102030405060708090A0B0C0D0E0F1011121314151617
        18191A1B1C1D1E1F2021222324252627
     P: 000102030405060708090A0B0C0D0E0F1011121314151617
        18191A1B1C1D1E1F2021222324252627
     C: D5CA91748410C1751FF8A2F618255B68A0A12E093FF45460
        6E59F9C1D0DDC54B65E8628E568BAD7AED07BA06A4A69483
        A7035490C5769E60

     N: BBAA9988776655443322110E
     A: 000102030405060708090A0B0C0D0E0F1011121314151617
        18191A1B1C1D1E1F2021222324252627
     P:
     C: C5CD9D1850C141E358649994EE701B68

     N: BBAA9988776655443322110F
     A:
     P: 000102030405060708090A0B0C0D0E0F1011121314151617
        18191A1B1C1D1E1F2021222324252627
     C: 4412923493C57D5DE0D700F753CCE0D1D2D95060122E9F15
        A5DDBFC5787E50B5CC55EE507BCB084E479AD363AC366B95
        A98CA5F3000B1479
```

Next are several internal values generated during the OCB-ENCRYPT computation for the last test vector listed above.

```
     L_*       : C6A13B37878F5B826F4F8162A1C8D879
     L_$       : 8D42766F0F1EB704DE9F02C54391B075
     L_0       : 1A84ECDE1E3D6E09BD3E058A8723606D
     L_1       : 3509D9BC3C7ADC137A7C0B150E46C0DA
     bottom    : 15 (decimal)
     Ktop      : 9862B0FDEE4E2DD56DBA6433F0125AA2
     Stretch   : 9862B0FDEE4E2DD56DBA6433F0125AA2FAD24D13A063F8B8
     Offset_0  : 587EF72716EAB6DD3219F8092D517D69
     Offset_1  : 42FA1BF908D7D8D48F27FD83AA721D04
     Offset_2  : 77F3C24534AD04C7F55BF696A434DDDE
     Offset_*  : B152F972B3225F459A1477F405FC05A7
     Checksum_1: 000102030405060708090A0B0C0D0E0F
     Checksum_2: 10101010101010101010101010101010
     Checksum_*: 30313233343536379010101010101010
```

The next tuple shows a result with a tag length of 96 bits and a different key.

```
  K: 0F0E0D0C0B0A09080706050403020100

  N: BBAA9988776655443322110D
  A: 000102030405060708090A0B0C0D0E0F1011121314151617
     18191A1B1C1D1E1F2021222324252627
  P: 000102030405060708090A0B0C0D0E0F1011121314151617
     18191A1B1C1D1E1F2021222324252627
  C: 1792A4E31E0755FB03E31B22116E6C2DDF9EFD6E33D536F1
     A0124B0A55BAE884ED93481529C76B6AD0C515F4D1CDD4FD
     AC4F02AA
```

The following algorithm tests a wider variety of inputs. Results are given for each parameter set defined in Section 3.1.

```
   K = zeros(KEYLEN-8) || num2str(TAGLEN,8)
   C = <empty string>
   for i = 0 to 127 do
      S = zeros(8i)
      N = num2str(3i+1,96)
      C = C || OCB-ENCRYPT(K,N,S,S)
      N = num2str(3i+2,96)
      C = C || OCB-ENCRYPT(K,N,<empty string>,S)
      N = num2str(3i+3,96)
      C = C || OCB-ENCRYPT(K,N,S,<empty string>)
   end for
   N = num2str(385,96)
   Output : OCB-ENCRYPT(K,N,C,<empty string>)
```

Iteration i of the loop adds 2i + (3 * TAGLEN / 8) bytes to C, resulting in an ultimate length for C of 22,400 bytes when TAGLEN == 128, 20,864 bytes when TAGLEN == 192, and 19,328 bytes when TAGLEN == 64. The final OCB-ENCRYPT has an empty plaintext component, so serves only to authenticate C. The output should be:

```
  AEAD_AES_128_OCB_TAGLEN128 Output: 67E944D23256C5E0B6C61FA22FDF1EA2
  AEAD_AES_192_OCB_TAGLEN128 Output: F673F2C3E7174AAE7BAE986CA9F29E17
  AEAD_AES_256_OCB_TAGLEN128 Output: D90EB8E9C977C88B79DD793D7FFA161C
  AEAD_AES_128_OCB_TAGLEN96 Output : 77A3D8E73589158D25D01209
  AEAD_AES_192_OCB_TAGLEN96 Output : 05D56EAD2752C86BE6932C5E
  AEAD_AES_256_OCB_TAGLEN96 Output : 5458359AC23B0CBA9E6330DD
  AEAD_AES_128_OCB_TAGLEN64 Output : 192C9B7BD90BA06A
  AEAD_AES_192_OCB_TAGLEN64 Output : 0066BC6E0EF34E24
  AEAD_AES_256_OCB_TAGLEN64 Output : 7D4EA5D445501CBE
```

Authors' Addresses

Ted Krovetz
Computer Science Department
California State University, Sacramento
6000 J Street
Sacramento, CA 95819-6021
USA

EMail: ted@krovetz.net

Phillip Rogaway
Computer Science Department
University of California, Davis
One Shields Avenue
Davis, CA 95616-8562
USA

EMail: rogaway@cs.ucdavis.edu