Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: January 16, 2012

T. Krovetz
Sacramento State
P. Rogaway
UC Davis

July 15, 2011

The OCB Authenticated-Encryption Algorithm
draft-krovetz-ocb-02

Abstract

This document specifies OCB, a shared-key blockcipher-based encryption scheme that provides privacy and authenticity for plaintexts and authenticity for associated data.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

Schemes for authenticated encryption (AE) simultaneously provide for privacy 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 privacy 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 privacy 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 [4]. 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 selected as a new value for each message encrypted. 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 privacy 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 and 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 on-line: 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 in the square of the number of blocks of texts divided by two to the blocklength.

The version of OCB defined in this document is a refinement of two prior schemes. The original OCB version was published in 2001 [6] and was listed as an optional component in IEEE 802.11i. A second version was published in 2004 [5] and is specified in ISO 19772. The scheme described here is called OCB3 in the 2011 paper describing the mode [4]; it shall be referred to simply as OCB throughout this document. See [4] for complete references, timing information, and a discussion of the differences between the algorithms.

2. Notation and Basic Operations

There are two types of variables used in this specification, strings and integers. Although most data processed by implementations of OCB will be byte-oriented, a number of bit-level operations are used in this specification, and so strings are here considered strings of bits rather than strings of bytes. String variables are always written with an initial upper-case letter while integer variables are written in all lower-case. 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 (eg, 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).

S[i..j]      The substring of S consisting of bits i through j,
             inclusive.

S || T       String S concatenated with string T (eg, 000 || 111 ==
             000111).

str2num(S)   The base-2 integral interpretation of bitstring S (eg,
             str2num(1110) == 14).

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.
```

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 [2].

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 [2].

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 [1]. 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.

4. OCB Algorithms

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

4.1. Associated-Data Processing: 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 an 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(P_*))) 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.

```
Function name:
  OCB-ENCRYPT
Input:
  K, string of KEYLEN bits                      // Key
  N, string of fewer than 128 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 an empty string.

  //
  // Nonce-dependent and per-encryption variables
  //
  Nonce = zeros(127-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 fewer than 128 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 an empty string.

     //
     // Nonce-dependent and per-decryption variables
     //
     Nonce = zeros(127-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, privacy and authenticity. Privacy 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 (N,C,T) triple 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 privacy 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 vanishes when s becomes as large as 2^{128/2}. 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.

It is crucial that, as one encrypts, one does not repeat a nonce. Repetition of a nonce will compromise both privacy and authenticity: partial information about past plaintexts will be revealed and subsequent forgeries will be possible. As a consequence, OCB must not be used in environemnts where the encrypting party cannot guarantee nonce uniqueness. Note that there are AEAD schemes, particularly SIV [3], appropriate for environements 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 should be partitioned so that no nonce that could be used by one party to encrypt could be used by the other to encrypt (eg, odd and even counters).

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 prefix bits of this internal tag are used for authentication. The length TAGLEN of the prefix used impacts the adversary's ability to forge: it will always be trivial for the adversary to forge with probability 2^{-TAGLEN}. It is up to the application designer to choose an appropriate value for TAGLEN. Longer tags cost no more computationally than do shorter ones.

Timing attacks are not a part of the formal security model and an implementation should take care to mitigate them. 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 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 allows ciphertexts to be parsed unambiguously.

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 Identifier
AEAD_AES_128_OCB_TAGLEN128	Section 3.1	XX
AEAD_AES_128_OCB_TAGLEN96	Section 3.1	XX
AEAD_AES_128_OCB_TAGLEN64	Section 3.1	XX
AEAD_AES_192_OCB_TAGLEN128	Section 3.1	XX
AEAD_AES_192_OCB_TAGLEN96	Section 3.1	XX
AEAD_AES_192_OCB_TAGLEN64	Section 3.1	XX
AEAD_AES_256_OCB_TAGLEN128	Section 3.1	XX
AEAD_AES_256_OCB_TAGLEN96	Section 3.1	XX
AEAD_AES_256_OCB_TAGLEN64	Section 3.1	XX

7. Acknowledgements

The design of the original OCB scheme [6] was done while Phil Rogaway was at Chiang Mai University, Thailand. Follow-up work [5] was done with support of NSF grant 0208842 and a gift from Cisco. The final work by Krovetz and Rogaway that has resulted in this spec [4] was supported by NSF grant 0904380.

8. References

8.1. Normative References

[1] McGrew, D., "An interface and algorithms for authenticated encryption", RFC 5116, January 2008.

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

8.2. Informative References

[3] Harkins, D., "Synthetic Initialization Vector (SIV) authenticated ancryption using the Advanced Encryption Standard (AES)", RFC 5297, October 2008.

[4] Krovetz, T. and P. Rogaway, "The software performance of authenticated-encryption modes", in Fast Software Encryption - FSE 2011, Springer, 2011.

[5] Rogaway, P., "Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC", in Advances in Cryptology - ASIACRYPT 2004, Springer, 2004.

[6] Rogaway, P., Bellare, M., Black, J., and T. Krovetz, "OCB: A block-cipher mode of operation for efficient authenticated encryption", in ACM Conference on Computer and Communications Security 2001 - CCS 2001, ACM Press, 2001.

Appendix A. Sample Results

This section gives sample output values for various inputs when using the AEAD_AES_128_OCB_TAGLEN128 parameters defined in Section 3.1. All strings are represented in hexadecimal (eg, 0F represents the bitstring 00001111).

Each of the following (A,P,C) triples show the ciphertext C that results from OCB-ENCRYPT(K,N,A,P) when K and N are fixed with the values

```
K : 000102030405060708090A0B0C0D0E0F
N : 000102030405060708090A0B
```

Empty entries indicate empty strings.

```
A:
P:
C: 197B9C3C441D3C83EAFB2BEF633B9182

A: 0001020304050607
P: 0001020304050607
C: 92B657130A74B85A16DC76A46D47E1EAD537209E8A96D14E

A: 0001020304050607
P:
C: 98B91552C8C009185044E30A6EB2FE21

A:
P: 0001020304050607
C: 92B657130A74B85A971EFFCAE19AD4716F88E87B871FBEED

A: 000102030405060708090A0B0C0D0E0F
P: 000102030405060708090A0B0C0D0E0F
C: BEA5E8798DBE7110031C144DA0B26122776C9924D6723A1F
   C4524532AC3E5BEB

A: 000102030405060708090A0B0C0D0E0F
P:
C: 7DDB8E6CEA68148662125096 19B19CC6

A:
```

```
P: 000102030405060708090A0B0C0D0E0F
C: BEA5E8798DBE7110031C144DA0B2612213CC8B747807121A
   4CBB3E4BD6B456AF

A: 000102030405060708090A0B0C0D0E0F1011121314151617
P: 000102030405060708090A0B0C0D0E0F1011121314151617
C: BEA5E8798DBE7110031C144DA0B26122FCFCEE7A2A8D4D48
   5FA94FC3F38820F1DC3F3D1FD4E55E1C

A: 000102030405060708090A0B0C0D0E0F1011121314151617
P:
C: 282026DA3068BC9FA118681D559F10F6

A:
P: 000102030405060708090A0B0C0D0E0F1011121314151617
C: BEA5E8798DBE7110031C144DA0B26122FCFCEE7A2A8D4D48
   6EF2F52587FDA0ED97DC7EEDE241DF68

A: 000102030405060708090A0B0C0D0E0F1011121314151617
   18191A1B1C1D1E1F
P: 000102030405060708090A0B0C0D0E0F1011121314151617
   18191A1B1C1D1E1F
C: BEA5E8798DBE7110031C144DA0B26122CEAAB9B05DF771A6
   57149D53773463CBB2A040DD3BD5164372D76D7BB6824240

A: 000102030405060708090A0B0C0D0E0F1011121314151617
   18191A1B1C1D1E1F
P:
C: E1E072633BADE51A60E85951D9C42A1B

A:
P: 000102030405060708090A0B0C0D0E0F1011121314151617
   18191A1B1C1D1E1F
C: BEA5E8798DBE7110031C144DA0B26122CEAAB9B05DF771A6
   57149D53773463CB4A3BAE824465CFDAF8C41FC50C7DF9D9

A: 000102030405060708090A0B0C0D0E0F1011121314151617
   18191A1B1C1D1E1F2021222324252627
P: 000102030405060708090A0B0C0D0E0F1011121314151617
   18191A1B1C1D1E1F2021222324252627
C: BEA5E8798DBE7110031C144DA0B26122CEAAB9B05DF771A6
   57149D53773463CB68C65778B058A635659C623211DEEA0D
   E30D2C381879F4C8

A: 000102030405060708090A0B0C0D0E0F1011121314151617
   18191A1B1C1D1E1F2021222324252627
P:
C: 7AEB7A69A1687DD082CA27B0D9A37096
```

```
A:
P: 000102030405060708090A0B0C0D0E0F1011121314151617
   18191A1B1C1D1E1F2021222324252627
C: BEA5E8798DBE7110031C144DA0B26122CEAAB9B05DF771A6
   57149D53773463CB68C65778B058A635060C8467F4ABAB5E
   8B3C2067A2E115DC
```

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

```
bottom    : 11
Checksum_1: 000102030405060708090A0B0C0D0E0F
Checksum_2: 10101010101010101010101010101010
Checksum_*: 30313233343536379010101010101010
Ktop      : 0000000100010203040506070809 0A00
L_*       : C6A13B37878F5B826F4F8162A1C8D879
L_$       : 8D42766F0F1EB704DE9F02C54391B075
L_0       : 1A84ECDE1E3D6E09BD3E058A8723606D
L_1       : 3509D9BC3C7ADC137A7C0B150E46C0DA
Offset_0  : 088A4C602C15FCCF8ECB3677E5E63517
Offset_1  : 120EA0BE322892C633F533FD62C5557A
Offset_2  : 270779020E524ED5498938E86C8395A0
Offset_*  : E1A6423589DD155726C6B98ACD4B4DD9
Stretch   : 43E111498C0582BF99F1D966CEFCBCC6A2F058C589873D26
```

The following pseudocode algorithm tests a wider variety of inputs. Results are given for each of AEAD_AES_128_OCB_TAGLEN128, AEAD_AES_192_OCB_TAGLEN128 and AEAD_AES_256_OCB_TAGLEN128. Let <i> be the 8-bit base-2 representation of i (eg, <3> == 00000011 and <255> == 11111111).

```
K = zeros(KEYLEN)           // Keylength of AES in use
for i = 0 to 127 do
   S = zeros(8i)            // i bytes of zeros
   N = zeros(88) || <i>     // 11 byte zero followed by 1 byte i
   C = C || OCB-ENCRYPT(K,N,S,S)
   C = C || OCB-ENCRYPT(K,N,<empty string>,S)
   C = C || OCB-ENCRYPT(K,N,S,<empty string>)
end for
N = zeros(96)
Output : OCB-ENCRYPT(K,N,C,<empty string>)
```

Iteration i of the loop adds 2i + 48 bytes to C, resulting in an ultimate length for C of 22,400 bytes. 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: B2B41CBF9B05037DA7F16C24A35C1C94
AEAD_AES_192_OCB_TAGLEN128 Output: 1529F894659D2B51B776740211E7D083
AEAD_AES_256_OCB_TAGLEN128 Output: 42B83106E473C0EEE086C8D631FD4C7B
```

Authors' Addresses

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

Email: ted@krovetz.net

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

Email: rogaway@cs.ucdavis.edu