Internet Engineering Task Force
Internet-Draft
Updates: 7523 (if approved)
Intended status: Informational
Expires: November 12, 2018

T. Krovetz
Sacramento State

May 11, 2018

OCB For Block Ciphers Without 128-Bit Blocks
draft-krovetz-ocb-wideblock-00

Abstract

The OCB authenticated-encryption algorithm is specified in RFC 7523, but only for blockciphers with 128-bit blocks such as AES. This document extends RFC 7523 by specifying how OCB is used with blockciphers of any blocklength. When the blocklength is 128 bits, this specification and that in RFC 7523 are identical.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on November 12, 2018.

Table of Contents

1. Introduction

OCB is a shared-key, blockcipher-based authentication scheme specified in [RFC7523]. It was designed with the AES blockcipher in mind and thus envisioned only being used with 128-bit blockciphers. The resulting RFC 7523 does not allow blockciphers with larger or smaller blocklengths. This document respecifies OCB in a more general manner, eliminating the expectation that a 128-bit blockcipher is used. This update is consistent with RFC 7523 and does not contradict it in any way. For applications using 128-bit blockciphers, RFC 7523 should be preferred because it is simpler, self-contained, and has more applicable test vectors.

Changing the blocklength used in OCB is not a simple matter. There are non-trivially defined constants used in OCB that must be recalculated for each different blocklength. What follows is largely

a copy of the algorithms from RFC 7523, made more general by using blocklength-dependent symbolic constants.

The security of OCB continues to follow a birthday bound. Both the confidentiality and the authenticity properties of OCB degrade as per s^2 / 2^b, where s is the total number of blocks that the adversary acquires and b is the number of bits per blockcipher block. Note that this means security degrades rapidly when using a blockcipher with a small to moderate blocklength.

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 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(001) == 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 (eg, 000 || 111 == 000111).

str2num(S)	The base-2 interpretation of bitstring S (eg, str2num(1110) == 14).
num2str(i,n)	The n-bit string whose base-2 interpretation is i (eg, num2str(14,4) == 1110 and num2str(1,2) == 01).
double(S)	If S[1] == 0 then double(S) == (S[2..bitlen(S)] || 0); otherwise double(S) == (S[2..bitlen(S)] || 0) xor num2str(RESIDUE,bitlen(S)) where RESIDUE is defined in Section 3.1 or Appendix B.

3. OCB Global Parameters

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

Specifying a blockcipher implicitly defines the following symbols.

BLOCKLEN	The blockcipher's blocklength, in bits.
KEYLEN	The blockcipher's key length, in bits.
ENCIPHER(K,P)	The blockcipher function mapping BLOCKLEN-bit plaintext block P to its corresponding ciphertext block using KEYLEN-bit key K.
DECIPHER(K,C)	The inverse blockcipher function mapping BLOCKLEN-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 positive value up to, and including, the smaller of BLOCKLEN or 256.

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

3.1. Constants Derived From BLOCKLEN

Each value of BLOCKLEN gives rise to constants that need careful choosing to ensure OCB security and efficiency. The value RESIDUE is used in the definition of double given in Section 2, and the values SHIFT, MASKLEN, and TAGREP are used in the OCB-ENCRYPT and OCB-DECRYPT functions given in Section 4.

The following table lists these constants for a collection of blockcipher blocklengths. If a blocklength is needed that is not in the table, Appendix B gives the criteria and algorithm used to determine the constants given here. The same criteria and algorithm should be used to generate other constants for other blocklengths.

Note that there are attacks on OCB with success proportional to a birthday bound related to BLOCKLEN. This means that using small values of BLOCKLEN may lead quickly to poor security. See Section 5 for more information on the security bounds of OCB.

BLOCKLEN	RESIDUE	SHIFT	MASKLEN	TAGREP
32	141	17	4	5
64	27	25	5	6
96	1601	33	6	7
128	135	8	6	7
192	135	40	7	8
256	1061	1	8	8
384	4109	80	8	8
512	293	176	8	8
768	655377	160	9	8
1024	524355	352	9	8
1600	18435	192	10	8

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. 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(BLOCKLEN) 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 BLOCKLEN bits                  // Hash result

   Sum is defined as follows.

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

     //
     // Consider A as a sequence of BLOCKLEN-bit blocks
     //
     Let m be the largest integer so that m * BLOCKLEN <= 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) == BLOCKLEN for each 1 <= i <= m.
       Note: A_* may possibly be the empty string.

     //
     // Process any whole blocks
     //
     Sum_0 = zeros(BLOCKLEN)
     Offset_0 = zeros(BLOCKLEN)
     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_*
        Zerofill = zeros(BLOCKLEN-(1+bitlen(A_*)))
        CipherInput = (A_* || 1 || Zerofill) 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 up to BLOCKLEN-(TAGREP+1) 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(BLOCKLEN))
  L_$ = double(L_*)
  L_0 = double(L_$)
  L_i = double(L_{i-1}) for every integer i > 0

  //
  // Consider P as a sequence of BLOCKLEN-bit blocks
  //
  Let m be the largest integer so that m * BLOCKLEN <= 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) == BLOCKLEN for each 1 <= i <= m.
    Note: P_* may possibly be the empty string.

  //
  // Nonce-dependent and per-encryption variables
  //
  Zerofill = zeros(BLOCKLEN-(TAGREP+1+bitlen(N)))
  Nonce = num2str(TAGLEN mod BLOCKLEN, TAGREP) || Zerofill || 1 || N
  bottom = str2num(Nonce[BLOCKLEN-MASKLEN+1..BLOCKLEN])
  Ktop = ENCIPHER(K, Nonce[1..BLOCKLEN-MASKLEN] || zeros(MASKLEN))
  ShiftedKtop = Ktop[1..BLOCKLEN-SHIFT] xor Ktop[1+SHIFT..BLOCKLEN]
  Stretch = Ktop || ShiftedKtop
  Offset_0 = Stretch[1+bottom..BLOCKLEN+bottom]
  Checksum_0 = zeros(BLOCKLEN)
```

```
    //
    // 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_*)]
       PaddedP = P_* || 1 || zeros(BLOCKLEN-(bitlen(P_*)+1))
       Checksum_* = Checksum_m xor PaddedP
       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 up to BLOCKLEN-(TAGREP+1) 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(BLOCKLEN))
L_$ = double(L_*)
L_0 = double(L_$)
L_i = double(L_{i-1}) for every integer i > 0

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

//
// Nonce-dependent and per-decryption variables
//
Zerofill = zeros(BLOCKLEN-(TAGREP+1+bitlen(N)))
Nonce = num2str(TAGLEN mod BLOCKLEN, TAGREP) || Zerofill || 1 || N
bottom = str2num(Nonce[BLOCKLEN-MASKLEN+1..BLOCKLEN])
Ktop = ENCIPHER(K, Nonce[1..BLOCKLEN-MASKLEN] || zeros(MASKLEN))
ShiftedKtop = Ktop[1..BLOCKLEN-SHIFT] xor Ktop[1+SHIFT..BLOCKLEN]
Stretch = Ktop || ShiftedKtop
Offset_0 = Stretch[1+bottom..BLOCKLEN+bottom]
Checksum_0 = zeros(BLOCKLEN)

//
// 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_*)]
   PaddedP = P_* || 1 || zeros(BLOCKLEN-bitlen(P_*)-1)
   Checksum_* = Checksum_m xor PaddedP
```

```
      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

What follows is a duplicate of the Security Considerations section of RFC 7523 with numeric literals replaced by their symbolic equivalents. With the ability to change BLOCKLEN comes two important additional considerations. The amount of data that can safely be processed using a single key is related to BLOCKLEN: a larger BLOCKLEN allows more data whereas a smaller BLOCKLEN allows less. Also, authentication tag lengths are limited to BLOCKLEN bits, which means that using OCB with a small BLOCKLEN forces the use of small tags. In all cases, but especially when reducing BLOCKLEN, a careful evaluation of an application's security requirements should be conducted and only if OCB, with a particular BLOCKLEN, meets those requirements should it be used.

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^BLOCKLEN, 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^(BLOCKLEN/2). Thus the user should never use a key to generate an amount of ciphertext that is near to, or exceeds, 2^(BLOCKLEN/2) blocks. In order to ensure that s^2 / 2^BLOCKLEN remains less than about 1/2^32, a given key should be used to encrypt at most 2^(BLOCKLEN/2-16) blocks, 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 BLOCKLEN-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 against 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 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 AD 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 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 (eg, odd and even counters).

6. IANA Considerations

The Internet Assigned Numbers Authority (IANA) has defined a registry for Authenticated Encryption with Associated Data parameters. This document does not specify any concrete AEAD schemes, so contributes nothing to the registry. Any AEAD scheme based on this document, where a permanently registered identifier would be useful, should register such identifier with IANA [RFC5116].

7. Acknowledgements

During a short period of 2017 three people inquired about extending OCB to blockciphers with blocklengths other than 128-bits. Thanks go to Jeffrey Walton, Mark Wooding, and Uri Blumenthal for providing the

motivation for this work. Mark Wooding has been especially helpful, providing the basis for the code in Appendix B and corroborating test vectors by writing an independent implementation.

8. References

8.1. Normative References

[Polys] Seroussi, G., "Table of low-weight binary irreducible polynomials", Hewlett-Packard technical report HPL-98-135, August 1998.

[RFC7523] Krovetz, T. and P. Rogaway, "The OCB authenticated-encryption algorithm", RFC 7523, May 2014.

8.2. Informative References

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

[RC6] Rivest, R., Robshaw, M., Sidney, R., and Y. Yin, "The RC6 block cipher", Posted on RSA Data Security website August 20, 1998.

[RFCRC6] Krovetz, T., "RC6 and RC5 test vectors for multiple block sizes", RFC XXXX, November 2018.

[RFC4107] Bellovin, S. and R. Housley, "Guidelines for cryptographic key management", RFC 4107, June 2005.

[RFC5116] McGrew, D., "An interface and algorithms for authenticated encryption", RFC 5116, January 2008.

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

[Sage] The Sage Developers, "SageMath, the Sage Mathematics Software System (Version 7.6)", DOI 10.5281/zenodo.820864, 2017, <http://www.sagemath.org/>.

Appendix A. Sample Results

This section gives sample outputs for various inputs using OCB and blockciphers with blocklengths other than 128. In each case, an instance of RC6 is used for the blockcipher [RC6]. The name RC6-w/r/

b refers to RC6 with a 4w-bit blocklength, r internal rounds, and a b-byte key.

There are too many possible cases to consider for this section to be exhaustive. Instead a small number of examples are shown in detail with many significant variable assignments in the pseudocode shown. This will help an implementor find most basic errors. Further validation could then be accomplished by comparison with a reference implementation over a much more thorough selection of inputs.

Each of the following entries indicates the blockcipher and tag length used followed by the values K, N, A, P, C, indicating that plaintext P with associated data A when encrypted using OCB with key K, and nonce N, yields ciphertext C (which includes the tag). All strings are represented in hexadecimal (eg, 0F represents the bitstring 00001111). An empty entry indicates the empty string.

A.1. Example 1: 64-bit Blockcipher, 0-byte Input

```
K: 000102030405060708090A0B0C0D0E0F
N: 000102030405
A:
P:
C: 474D57A4043E

Assignments during HASH(K,A)

L_*: 39EF0C3FF4475894
L_$: 73DE187FE88EB128
Sum: 0000000000000000

Assignments during OCB-ENCRYPT

L_*: 39EF0C3FF4475894
L_$: 73DE187FE88EB128
Nonce: C001000102030405
bottom: 5
Ktop: 8298E905914FB488
Stretch: 8298E905914FB48889BA766C814FB488
Offset_0: 531D20B229F69111
Tag: 474D57A4043E7768
```

A.2. Example 2: 64-bit Blockcipher, 4-byte Input

```
RC6-16/16/16 (64-bit blocks), 48-bit tags
K: 000102030405060708090A0B0C0D0E0F
N: 000102030405
A: 00010203
P: 00010203
C: B75BF470BB0B354C5B92

Assignments during HASH(K,A)

L_*: 39EF0C3FF4475894
L_$: 73DE187FE88EB128
Offset_*: 39EF0C3FF4475894
CipherInput: 39EE0E3C74475894
Sum: 48068A4FD12ABB4C

Assignments during OCB-ENCRYPT

L_*: 39EF0C3FF4475894
L_$: 73DE187FE88EB128
Nonce: C001000102030405
bottom: 5
Ktop: 8298E905914FB488
Stretch: 8298E905914FB48889BA766C814FB488
Offset_0: 531D20B229F69111
Offset_*: 6AF22C8DDDB1C985
Pad: B75AF6732D339D4B
PaddedP: 0001020380000000
Checksum_*: 0001020380000000
C_*: B75BF470
Tag: BB0B354C5B92CBAA
```

A.3. Example 3: 64-bit Blockcipher, 8-byte Input

RC6-16/16/16 (64-bit blocks), 48-bit tags
K: 000102030405060708090A0B0C0D0E0F
N: 000102030405
A: 0001020304050607
P: 0001020304050607
C: 882A4FFE11B0E540D41F335172BD

Assignments during HASH(K,A)

L_*: 39EF0C3FF4475894
L_$: 73DE187FE88EB128
L_0: E7BC30FFD11D6250
Offset_1: E7BC30FFD11D6250
Sum_1: E3549C18B6F398EB
Sum: E3549C18B6F398EB

Assignments during OCB-ENCRYPT

L_*: 39EF0C3FF4475894
L_$: 73DE187FE88EB128
Nonce: C001000102030405
bottom: 5
Ktop: 8298E905914FB488
Stretch: 8298E905914FB48889BA766C814FB488
Offset_0: 531D20B229F69111
L_0: E7BC30FFD11D6250
Offset_1: B4A1104DF8EBF341
Checksum_1: 0001020304050607
C_1: 882A4FFE11B0E540
Tag: D41F335172BD43B7

A.4. Example 4: 64-bit Blockcipher, 20-byte Input

RC6-16/16/16 (64-bit blocks), 48-bit tags
K: 000102030405060708090A0B0C0D0E0F
N: 000102030405
A: 000102030405060708090A0B0C0D0E0F10111213
P: 000102030405060708090A0B0C0D0E0F10111213
C: 882A4FFE11B0E540F218C902D76A601F90DF1E27BD403C47DF4E

Assignments during HASH(K,A)

L_*: 39EF0C3FF4475894
L_$: 73DE187FE88EB128
L_0: E7BC30FFD11D6250
Offset_1: E7BC30FFD11D6250
Sum_1: E3549C18B6F398EB
L_1: CF7861FFA23AC4BB
Offset_2: 28C451007327A6EB
Sum_2: FADB0507B1DA1433
Offset_*: 112B5D3F8760FE7F
CipherInput: 013A4F2C0760FE7F
Sum: D761B0AFAA584CF7

Assignments during OCB-ENCRYPT

L_*: 39EF0C3FF4475894
L_$: 73DE187FE88EB128
Nonce: C001000102030405
bottom: 5
Ktop: 8298E905914FB488
Stretch: 8298E905914FB48889BA766C814FB488
Offset_0: 531D20B229F69111
L_0: E7BC30FFD11D6250
Offset_1: B4A1104DF8EBF341
Checksum_1: 0001020304050607
C_1: 882A4FFE11B0E540
L_1: CF7861FFA23AC4BB
Offset_2: 7BD971B25AD137FA
Checksum_2: 0808080808080808
C_2: F218C902D76A601F
Offset_*: 42367D8DAE966F6E
Pad: 80CE0C34E04B0990
PaddedP: 1011121380000000
Checksum_*: 18191A1B88080808
C_*: 90DF1E27
Tag: BD403C47DF4ED926

A.5. Example 5: 256-bit Blockcipher, 80-byte Input

```
RC6-64/16/16 (256-bit blocks), 256-bit tags
K: 000102030405060708090A0B0C0D0E0F
N: 000102030405060708090A0B
A: 000102030405060708090A0B0C0D0E0F
   101112131415161718191A1B1C1D1E1F
   202122232425262728292A2B2C2D2E2F
   303132333435363738393A3B3C3D3E3F
   404142434445464748494A4B4C4D4E4F
P: 000102030405060708090A0B0C0D0E0F
   101112131415161718191A1B1C1D1E1F
   202122232425262728292A2B2C2D2E2F
   303132333435363738393A3B3C3D3E3F
   404142434445464748494A4B4C4D4E4F
C: 5327CF07146F7B51A5D7AABA93D7F626
   2F65ED3931815A5EF681217EBBF2BB13
   F464772416E08D4BEAC37DFD8F614D98
   AD7D58B1A63A24BB1CFE491D02AE28ED
   BC6D07CF935C31FC81361EA46FBF568F
   E4E72EE35BBD8FBBD04B2FCDB9656044
   A9EFE1140327CEBB9A1750CDBD1BD3EC

Assignments during HASH(K,A)

L_*: 6E75A413F50216C512AD330BFABE641B
     50E88C29BE5980AA2A09E43990125CBB
L_$: DCEB4827EA042D8A255A6617F57CC836
     A1D118537CB301545413C8732024B976
L_0: B9D6904FD4085B144AB4CC2FEAF9906D
     43A230A6F96602A8A82790E6404976C9
Offset_1: B9D6904FD4085B144AB4CC2FEAF9906D
          43A230A6F96602A8A82790E6404976C9
Sum_1: B1836672FE03B925800743FF4DE116B3
       C09C00B396CB20B058FAD1AEC4171F57
L_1: 73AD209FA810B6289569985FD5F320DA
     8744614DF2CC0551504F21CC8092E9B7
Offset_2: CA7BB0D07C18ED3CDFDD54703F0AB0B7
          C4E651EB0BAA07F9F868B12AC0DB9F7E
Sum_2: C16F97D1E09FA431655057CF582E8F32
       FAE7971E00DB1EA56E7BA4AC493ECD8F
Offset_*: A40E14C3891AFBF9CD70677BC5B4D4AC
          940EDDC2B5F38753D261551350C9C3C5
CipherInput: E44F5680CD5FBDBE85392D3089F99AE3
             140EDDC2B5F38753D261551350C9C3C5
Sum: 964A5EA730124E1EA6CD7874A58F449C
     5C3B2ED878115CC292A021FAF261B4FD
```

Assignments during OCB-ENCRYPT

```
L_*: 6E75A413F50216C512AD330BFABE641B
     50E88C29BE5980AA2A09E43990125CBB
L_$: DCEB4827EA042D8A255A6617F57CC836
     A1D118537CB301545413C8732024B976
Nonce: 000000000000000000000000000000000
       000000010001020304050607080 90A0B
bottom: 11
Ktop: 67DB009692E6C7CCEFBFE4F9B810544E
      DBD469326F0CBA90F94D02A6A85C7C9B
Stretch: 67DB009692E6C7CCEFBFE4F9B810544E
         DBD469326F0CBA90F94D02A6A85C7C9B
         A86D01BBB72B485530C02D0AC830FCD3
         6C7CBB56B115CFB10BD707EBF8E485AD
Offset_0: D804B497363E677DFF27CDC082A276DE
          A349937865D487CA68153542E3E4DD43
L_0: B9D6904FD4085B144AB4CC2FEAF9906D
     43A230A6F96602A8A82790E6404976C9
Offset_1: 61D224D8E2363C69B59301EF685BE6B3
          E0EBA3DE9CB28562C032A5A4A3ADAB8A
Checksum_1: 000102030405060708090A0B0C0D0E0F
            101112131415161718191A1B1C1D1E1F
C_1: 5327CF07146F7B51A5D7AABA93D7F626
     2F65ED3931815A5EF681217EBBF2BB13
L_1: 73AD209FA810B6289569985FD5F320DA
     8744614DF2CC0551504F21CC8092E9B7
Offset_2: 127F04474A268A4120FA99B0BDA8C669
          67AFC2936E7E8033907D8468233F423D
Checksum_2: 20202020202020202020202020202020
            20202020202020202020202020202020
C_2: F464772416E08D4BEAC37DFD8F614D98
     AD7D58B1A63A24BB1CFE491D02AE28ED
Offset_*: 7C0AA054BF249C843257AABB4716A272
          37474EBAD0270099BA746051B32D1E86
Pad: FC2C458CD71977BBC97F54EF23F218C0
     7C6AD1938FDE8374AF27960469595171
PaddedP: 404142434445464748494A4B4C4D4E4F
         80000000000000000000000000000000
Checksum_*: 606162636465666768696A6B6C6D6E6F
            A020202020202020202020202020202020
C_*: BC6D07CF935C31FC81361EA46FBF568F
Tag: E4E72EE35BBD8FBBD04B2FCDB9656044
     A9EFE1140327CEBB9A1750CDBD1BD3EC
```

A.6. Example 6: Extended Tests

The following VALIDATE[B] algorithm tests a wider variety of inputs. Parameter B indicates that throughout the validation RC6-(B/4)/16/16 (ie, RC6 with B-bit blocklength, 16 rounds, and 16-byte keys) is the blockcipher used. Please note that when B is not a power of two, then the non-standard version of RC6 documented in [RFCRC6] is used. In all cases TAGLEN is min(B, 256).

```
Function name:
  VALIDATE[B]
Output:
  Y, string of length min(B, 256) bits
```

Y is defined as follows.

```
   K = 000102030405060708090A0B0C0D0E0F
   C = <empty string>
   for i = 0 to 127 do
      S = (0x00 || 0x01 || 0x02 || ...)[1..8i]
      N = num2str(3i+1,16)
      C = C || OCB-ENCRYPT(K,N,S,S)
      N = num2str(3i+2,16)
      C = C || OCB-ENCRYPT(K,N,<empty string>,S)
      N = num2str(3i+3,16)
      C = C || OCB-ENCRYPT(K,N,S,<empty string>)
   end for
   N = num2str(385,16)
   Y = 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 16256 + 48 * TAGLEN bytes. The final OCB-ENCRYPT has an empty plaintext component, so serves only to authenticate C. The output values should be:

```
VALIDATE[32]   == 5A126BD4
VALIDATE[64]   == 21CE70BE54BDD72D
VALIDATE[96]   == 1014F0D84D8060DC47752BAA
VALIDATE[128]  == 7F9A12DE01C2C3150EBB2593D6531EA4
VALIDATE[192]  == 9A078741E919C04D27D788225F4EDC99
                  EB0864E3AA36C194
VALIDATE[256]  == 4D40016D7A255F603110AF8157863D4C
                  C392A2A2026C3CADF275583659389A84
VALIDATE[384]  == 64973A1E77815999A50E9B2A0BAFE802
                  F00F821D5B34E1F63E0D4DF6BA50F600
VALIDATE[512]  == 2CE8DA2D843D7373FA138C663305EC25
                  88FBCCE30DE426E99644E777F3FBCFD5
VALIDATE[768]  == 37848B8A255C73CE0EE3BE5C8277340C
                  BE64B54178B8731E052ECEC6BE9E5A07
VALIDATE[1024] == 439667F8BFFFC9D4C16E5CB3DE921F7D
                  AD0B76463AB56321EDB68F2D1D2AEEC0
VALIDATE[1600] == 794CCD79BC11854275C924DF56CB82F2
                  C443631EF9F9F90251DC6DDC2CCF7F08
```

Appendix B. Generating RESIDUE, SHIFT, MASKLEN, TAGREP Constants

OCB, as defined in [RFC7523] and [OCB], uses several "magic numbers" when manipulating 128-bit blocks (135, 8, 6 and 7 for RESIDUE, SHIFT, MASKLEN and TAGREP). These constants are carefully chosen and depend on BLOCKLEN. A table in Section 3.1 gives values for other assorted blocklengths and this section describes how to produce the values for more blocklengths if needed.

Finding MASKLEN and SHIFT for a particular BLOCKLEN begins by following the process described in Section 4.1 of [OCB]. The result of that process is the domain size for every possible shift, 1 through BLOCKLEN-1, in the stretch-then-shift hash used internally by OCB. With that information, MASKLEN and SHIFT are determined as follows. Define MASKLEN as the floor of the base-2 logarithm of the largest domain size found. Keep as SHIFT candidates all shifts with domain sizes of at least 2^MASKLEN. These all provide equal security, but we want a shift that allows efficient implementation too. A shift equal to a large power-of-two would be ideal, but not all BLOCKLEN values produce candidate shifts with that property. Of all the shifts with domains of at least 2^MASKLEN, we continue to keep as candidates those shift values that are minimal modulo 8 (if this minima is equal to zero, then shifts can be performed using byte manipulations). Finally, we prefer shifts that are close to a large power-of-two, so of the remaining candidates, we choose for SHIFT one whose value modulo 8 and modulo 2^k is equal for the largest k, and if more than one has this property we settle on the smallest of those that do. The Sage script at the end of this section makes explicit this process and was run using SageMath 7.6 to generate all of the

MASKLEN and SHIFT values given in Section 3.1 [Sage]. Its list of blocklengths can be edited and the script rerun to determine other MASKLEN and SHIFT values.

For the other two constants, TAGREP is defined as min(8, t) where t is the smallest integer making 2^t >= BLOCKLEN, and RESIDUE is extracted from [Polys]. To determine a RESIDUE for BLOCKLEN, find the entry whose largest number is BLOCKLEN and let S be the set of numbers other than BLOCKLEN in this entry. Then, RESIDUE = 1 + sum({2^x | x in S}). For example, when BLOCKLEN is 122 we find the entry "122,6,2,1" making S == {6,2,1}. In this case RESIDUE == 1 + 2^6 + 2^2 + 2^1 == 71.

B.1. Sage Script For Generating MASKLEN and SHIFT

```
<CODE BEGINS>
# w will iterate over these blocklengths. Edit list as you wish.
for w in [32, 64, 96, 128, 192, 256, 384, 512, 768, 1024, 1600]:

   domsize = [0]    # domsize[i] will contain domain size of shift i

   # Find domain size for each possible shift c in 1..w-1
   for c in range(1, w):

      # Build A matrix as specified in Section 4.1 of [OCB]
      I_rows = [[j == i for j in range(w)] for i in range(w)]
      J_rows = [[j == i or j == i + c for j in range(w)]
                                      for i in range(w)]
      IJ = matrix(GF(2), I_rows + J_rows)
      A = [IJ[i:i + w, 0:w] for i in range(w)]

      # Find number of qualifying sub-matrices (ie, domain size)
      i = 0         # increase i until not full-rank
      dom = w       # Set dom=i ends loop & sets dom to domain size
      while i < dom:
         if A[i].rank() < w: dom=i
         j = 0
         while j < i and i < dom:
            if (A[i] + A[j]).rank() < w: dom=i
            j = j + 1
         i = i + 1
      domsize.append(dom)   # set domsize[c] = dom

   # Generate shifts that are secure, in preference order
   domain_bits = floor(log(max(domsize), 2))
   candidates = (k for i in range(8)
                   for j in range(floor(log(w, 2)),2,-1)
                   for k in range(i,w,2**j)
                   if domsize[k] >= 2**domain_bits)

   # Print the first one found, or -1 if none
   print("block bits: %d, mask bits: %d, shift bits: %d" %
            (w, domain_bits, next(candidates,-1)))
<CODE ENDS>
```

Author's Address

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

Email: ted@krovetz.net