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

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 http://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 July 23, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ........................................... 2
2. Notation and Basic Operations .......................... 3
3. OCB Global Parameters .................................. 4
   3.1. Named OCB Parameter Sets and RFC 5116 Constants .... 4
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. 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 [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. See [OCB3] 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.

$\text{bitlen}(S)$ The length of string $S$ in bits (e.g., $\text{bitlen}(101) == 3$).

$\text{zeros}(n)$ 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 [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].

---

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.

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(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.
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_*
Note: P_* may possibly be the 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_*)))
else
C_* = <empty string>
Tag = ENCIPHER(K, Checksum_* xor Offset_* 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 the 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. To ensure these limits are not crossed, automated key management is recommended in systems exchanging large amounts of data [RFC4107].
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 he 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).

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 used for authentication. The value of TAGLEN impacts the adversary’s ability to forge: it will always be trivial for the adversary to forge with probability $2^{-\text{TAGLEN}}$. It is up to the application designer to choose an appropriate value for TAGLEN. Long tags cost no more computationally than short ones.

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 $(C_1 || C_2 || ... || C_m || C_*, \text{Tag}[1..\text{TAGLEN}])$ instead of the concatenation $C_1 || C_2 || ... || C_m || C_* || \text{Tag}[1..\text{TAGLEN}]$ introduces no security concerns. Because TAGLEN is fixed, both versions allow 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Numeric Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>AEAD_AES_128_OCB_TAGLEN128</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
<tr>
<td>AEAD_AES_128_OCB_TAGLEN96</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
<tr>
<td>AEAD_AES_128_OCB_TAGLEN64</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
<tr>
<td>AEAD_AES_192_OCB_TAGLEN128</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
<tr>
<td>AEAD_AES_192_OCB_TAGLEN96</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
<tr>
<td>AEAD_AES_192_OCB_TAGLEN64</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
<tr>
<td>AEAD_AES_256_OCB_TAGLEN128</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
<tr>
<td>AEAD_AES_256_OCB_TAGLEN96</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
<tr>
<td>AEAD_AES_256_OCB_TAGLEN64</td>
<td>Section 3.1</td>
<td>XX</td>
</tr>
</tbody>
</table>

7. Acknowledgements

The design of the original OCB scheme [OCB1] was done while Phil 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 spec was supported by NSF grant 0904380. Thanks go to the Crypto Forum Research Group for providing feedback on earlier drafts, and David McGrew for contributing some text in Section 5.

8. References

8.1. Normative References


8.2. Informative References


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

An empty entry indicates the empty string.
A:
P:
C: 197B9C3C441D3C83EAFB2BEF633B9182

A: 0001020304050607
P: 0001020304050607
C: 92B657130A74B85A16DC76A46D47E1EAD537209E8A96D14E

A: 0001020304050607
P:
C: 98B91552C8C009185044E30A6EB2FE21

A:
P:
C: 92B657130A74B85A971EFFCAE19AD4716F88E87B871FBEED

A: 000102030405060708090A0B0C0D0E0F
P: 000102030405060708090A0B0C0D0E0F
C: BEA5E879DBE711031C144DA0B26122776C9924D6723A1F
   C4524532AC3E5BEB

A: 000102030405060708090A0B0C0D0E0F
P:
C: 7DD8E6CEA6814866212509619B19CC6

A:
P:
C: BEA5E879DBE711031C144DA0B26122776C9924D6723A1F
   4CBB3E4BD6B456AF

A: 000102030405060708090A0B0C0D0E0F1011121314151617
P: 000102030405060708090A0B0C0D0E0F1011121314151617
C: BEA5E879DBE711031C144DA0B26122776C9924D6723A1F
   5FA94FC3F38820F1DC3F3D1FD4E55E1C

A: 000102030405060708090A0B0C0D0E0F1011121314151617
P:
C: 282026DA3068BC9FA118681D559F10F6

A:
P:
C: BEA5E879DBE711031C144DA0B26122776C9924D6723A1F
   6E6F2F52587FDA0ED97DC7EED2E241DF68

A: 000102030405060708090A0B0C0D0E0F1011121314151617
18191A1B1C1D1E1F
P: 000102030405060708090A0B0C0D0E0F1011121314151617
18191A1B1C1D1E1F
C: BEA5E879DBE711031C144DA0B26122776C9924D6723A1F
   5714D5373463CBB2A040DD3BD5164372D76D7BB6824240

A: 000102030405060708090A0B0C0D0E0F1011121314151617
18191A1B1C1D1E1F
The following 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 1 (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