BS ISO/IEC 9797-3:2011+A1:2020

BSI Standards Publication

Information technology — Security techniques — Message Authentication Codes (MACs)

Part 3: Mechanisms using a universal hash-function

National foreword

This British Standard is the UK implementation of ISO/IEC 9797-3:2011+A1:2020. It supersedes BS ISO/IEC 9797-3:2011, which is withdrawn.

The start and finish of text introduced or altered by amendment is indicated in the text by tags. Tags indicating changes to ISO/IEC text carry the number of the ISO/IEC amendment. For example, text altered by ISO/IEC amendment 1 is indicated by [A1⟩ ⟨A1].

The UK participation in its preparation was entrusted to Technical Committee IST/33/2, Cryptography and Security Mechanisms.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

Published by BSI Standards Limited 2020

ISBN 978 0 539 02649 8

ICS 35.030

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 January 2012.

Amendments/corrigenda issued since publication

Date	Text affected
29 February 2020	Implementation of ISO/IEC amendment A1:2020

INTERNATIONAL STANDARD

ISO/IEC 9797-3

First edition
2011-11-15

AMENDMENT 1
2020-02

Information technology — Security techniques — Message Authentication Codes (MACs) —

Part 3: **Mechanisms using a universal hash-function**

Technologies de l'information — Techniques de sécurité — Codes d'authentification de message (MAC) —

Partie 3: Mécanismes utilisant une fonction de hachage universelle

Reference number
ISO/IEC 9797-3:2011(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 9797-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 27, *IT Security techniques*.

ISO/IEC 9797 consists of the following parts, under the general title *Information technology — Security techniques — Message Authentication Codes (MACs)*:

— *Part 1: Mechanisms using a block cipher*

— *Part 2: Mechanisms using a dedicated hash-function*

— *Part 3: Mechanisms using a universal hash-function*

Introduction

In an IT environment, it is often required that one can verify that electronic data has not been altered in an unauthorized manner and that one can provide assurance that a message has been originated by an entity in possession of the secret key. A MAC (Message Authentication Code) algorithm is a commonly used data integrity mechanism that can satisfy these requirements.

This part of ISO/IEC 9797 specifies four MAC algorithms using universal hash-functions: UMAC, Badger, Poly1305-AES and GMAC.

These mechanisms can be used as data integrity mechanisms to verify that data has not been altered in an unauthorized manner. They can also be used as message authentication mechanisms to provide assurance that a message has been originated by an entity in possession of the secret key. The strength of the data integrity mechanism and message authentication mechanism is dependent on the length (in bits) and secrecy of the key, on the length (in bits) of a hash-code produced by the hash-function, on the strength of the hash-function, on the length (in bits) of the MAC, and on the specific mechanism.

NOTE A general framework for the provision of integrity services is specified in ISO/IEC 10181-6[7].

This page deliberately left blank

Information technology — Security techniques — Message Authentication Codes (MACs) —

Part 3: Mechanisms using a universal hash-function

1 Scope

This part of ISO/IEC 9797 specifies the following MAC algorithms that use a secret key and a universal hash-function with an n-bit result to calculate an m-bit MAC based on the block ciphers specified in ISO/IEC 18033-3 and the stream ciphers specified in ISO/IEC 18033-4:

a) UMAC;

b) Badger;

c) Poly1305-AES;

d) GMAC.

2 Normative references

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

ISO/IEC 9797-1, *Information technology — Security techniques — Message Authentication Codes (MACs) — Part 1: Mechanisms using a block cipher*

ISO/IEC 18031, *Information technology — Security techniques — Random bit generation*

ISO/IEC 18033-3, *Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers*

ISO/IEC 18033-4, *Information technology — Security techniques — Encryption algorithms — Part 4: Stream ciphers*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 9797-1 and the following apply.

3.1
empty string
string of symbols of length zero

3.2
key
sequence of symbols that controls the operation of a cryptographic transformation

3.3
nonce
number used once

3.4
prime number
positive integer greater than 1 which has no integer divisors other than 1 and itself

3.5
tag
result of a MAC algorithm, adjoined to a possibly encrypted message to provide integrity protection

3.6
universal hash-function
function mapping strings of bits to fixed-length strings of bits, indexed by a parameter called the key, satisfying the property that for all distinct inputs, the probability over all keys that the outputs collide is small

Note 1 to entry: Universal hash-functions were introduced by Carter and Wegman[4], and their application in MAC algorithms was first described by Wegman and Carter [10].

4 Symbols and abbreviated terms

The following notation is used in this part of ISO/IEC 9797:

bit(S,n) Returns the integer 1 if the n^{th} bit of the string S is 1, otherwise returns the integer 0 (indices begin at 1).

bitlength(S) Length of a string S in bits.

bitstr2uint(S) The non-negative integer whose binary representation is the string S. More formally, if S is t bits long then bitstr2uint(S) = 2^{t-1} * bit(S,1) + 2^{t-2} * bit(S,2) + ... + 2^1 * bit(S,t-1) + bit(S,t).

NOTE Bit strings are treated big-endian, i.e. the first bit is the most significant.

blocklen Block length of the underlying block cipher in octets.

ceil Rounding-up operation, i.e. if x is a floating-point number, then ceil(x) is the smallest integer n with $n \geq x$.

Enc(K, X) Encryption of a plaintext block X under a key K using a block cipher Enc.

floor Rounding-down operation, i.e. if x is a floating-point number, then floor(x) is the largest integer n with $n \leq x$.

H Hash value.

K Master key.

K_E Encryption key.

K_H Hash key.

keylen Block cipher key length in octets.

$\log_2$ Binary logarithm function.

M Message.

MAC Message authentication code.

max Largest value amongst those given as argument.

N Nonce.

octetlength(*S*) Length of a string *S* in octets (where *S* is assumed to have bitlength a multiple of 8).

octetstr2uint(*S*) The non-negative integer defined as $S[0] + 2^8 * S[1] + 2^{16} * S[2] + ... + 2^{8n-8} * S[n-1]$, where n = octetlength(*S*).

NOTE Octet strings are treated little-endian, i.e. the first octet is the least significant.

prime(*n*) Largest prime number smaller than 2^n, for any positive integer *n*.

NOTE The prime numbers used in this part of ISO/IEC 9797 are listed in Table 1.

Table 1 — Prime numbers

n	prime(*n*)	prime(*n*) in hexadecimal format					
32	$2^{32} - 5$					0x	FFFFFFFB
36	$2^{36} - 5$				0x	0000000F	FFFFFFFB
64	$2^{64} - 59$				0x	FFFFFFFF	FFFFFFC5
128	$2^{128} - 159$		0x	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFF61
130	$2^{130} - 5$	0x	00000003	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFB

$S[i]$ The *i*-th octet of the string *S* (indices begin at 0).

NOTE The specification for UMAC in 6.2 uses a starting index of 1 rather than 0.

$S[i...j]$ The substring of *S* consisting of octets *i* through *j*.

taglen Length of the tag, in octets.

uint2bitstr(*x*,*n*) The *n*-octet string *S* such that bitstr2uint(*S*) = *x*.

uint2octetstr(*x*,*n*) The *n*-octet string *S* such that *x* = octetstr2uint(*S*).

$X|_s$ Left-truncation of the block of bits *X*: if *X* has bit-length greater than or equal to *s*, then $X|_s$ is the *s*-bit block consisting of the left-most *s* bits of *X*.

$X|^s$ Right-truncation of the block of bits *X*: if *X* has bit-length greater than or equal to *s*, then $X|^s$ is the *s*-bit block consisting of the right-most *s* bits of *X*.

X>>1 Right shift of a block of bits *X* by one position: the leftmost bit of *Y* = *X*>>1 will always be set to zero.

|X| The length of *X* in bits.

zeropad(*S*,*n*) For positive integer *n*, the string *S* is padded with zero-bits to the nearest positive multiple of *n* octets. Formally, zeropad(*S*,*n*) = *S* || *T*, where T is the shortest string of zero-bits (possibly empty) so that *S* || T is non-empty and *n* divides octetlength(*S* || *T*).

$\oplus$ Bit-wise exclusive-OR operation on bit-strings. If *A*, *B* are strings of the same length then $A \oplus B$ is the string equal to the bit-wise logical exclusive-OR of *A* and *B*.

$\wedge$ Bit-wise logical AND operation on bit-strings. If *A*, *B* are strings of the same length then $A \wedge B$ is the string equal to the bit-wise logical AND of *A* and *B*.

$+_{32}$ Addition of two 32-bit strings, resulting in a 32-bit string. More formally, $S +_{32} T$ = uint2bitstr(bitstr2uint(*S*) + bitstr2uint(*T*) mod 2^{32}, 4).

$+_{64}$ Addition of two 64-bit strings, resulting in a 64-bit string. More formally, $S +_{64} T$ = uint2bitstr(bitstr2uint(S) + bitstr2uint(T) mod 2^{64}, 8).

$*$ Multiplication operator on integers.

$*_{64}$ Multiplication of two 64-bit strings, resulting in a 64-bit string. More formally, $S *_{64} T$ = uint2bitstr(bitstr2uint(S) * bitstr2uint(T) mod 2^{64}, 8).

NOTE The operations $+_{32}$, $+_{64}$ and $*_{64}$ correspond well with the addition and multiplication operations that are performed efficiently by modern computers.

|| Concatenation of two bit strings. If A and B are bit strings of lengths a and b respectively, then $A \| B$ is the bit string of length $a+b$ whose left most (first) a bits are the bits of A, and whose rightmost (last) b bits are the bits of B.

0^n String consisting of n zero-bits.

1^n String consisting of n one-bits.

{ } A bit-string with zero length.

• Multiplication in the field GF(2^{128}). The defining polynomial that determines the representation of GF(2^{128}) is $1 + \alpha + \alpha^2 + \alpha^7 + \alpha^{128}$.

NOTE Let U and V be 128-bit blocks. Then the 128-bit block $W = U \bullet V$ can be computed as follows:

a) Let $W = 0^{128}$ and $Z = U$.

b) For i = 1, 2, …, 128, perform the following two steps:

  1) If bit(V,i) = 1 then let $W = W \oplus Z$;

  2) If bit(Z,128) = 0 then let Z = Z>>1; otherwise let Z = (Z>>1) $\oplus$ (11100001 || 0^{120}).

Variables in capital letters denote strings; variables in small letters are integers.

5 General model

Message authentication codes based on universal hashing makes use of an encryption algorithm (block cipher or stream cipher). This type of message authentication codes has the special property that their security can be proven under the assumption that the encryption algorithm is secure.

MAC algorithms based on universal hashing require a master key K, a message M and a nonce value N as input. A MAC is computed using the following sequence of steps:

1) Key preprocessing. The master key K is used to generate a hash key K_H and an encryption key K_E.

2) Message preprocessing. The input message M is encoded into the necessary input format for the hash-function.

3) Message hashing. The encoded message is hashed under the control of the hash key K_H, using a universal hash-function. The result is a hash value H of fixed, short length.

4) Finalization. The hash value H is encrypted under the control of the encryption key K_E. The result is the message authentication code MAC.

For all mechanisms presented in this part of ISO/IEC 9797, the length of the input message is expected to consist of an integer number of octets.

NOTE For all universal-hash based MAC algorithms, it is of utmost importance that a different nonce is used for each new message that is authenticated under the same key. If this security requirement is not met, the security of the algorithm is severely reduced.

[A1) Annex A defines object identifiers that shall be used to identify the algorithms specified in this document. Annex B provides numerical examples for the algorithms specified in this document, and Annex C gives information on the security properties of these algorithms. (A1]

6 Mechanisms

6.1 Introduction

In this clause, four mechanisms using a universal hash-function are specified.

6.2 UMAC

6.2.1 Description of UMAC

UMAC is a family of four MAC algorithms optimized for different output bit-lengths, denoted by UMAC-32, UMAC-64, UMAC-96, and UMAC-128. UMAC can be used with any block cipher from ISO/IEC 18033-3. If the block cipher used has key length $|K|$ bits and block length $|B|$ bits, then UMAC uses a $|K|$-bit key K, and the length of the nonce N is between 8 and $|B|$ bits. Depending on which member of the UMAC family is used, the length of the MAC produced is 32, 64, 96, or 128 bits. This is represented by the parameter *taglen*, which can be 4, 8, 12 or 16 octets, respectively. The length of the input message shall be less than 2^{67} octets. The message input to the UMAC function shall contain a whole number of octets, i.e. the bitlength shall be a multiple of 8. If the bitlength is not a multiple of 8, this mechanism shall not be used.

NOTE 1 The version of UMAC specified here must not be confused with earlier versions of the UMAC algorithm, e.g.[2].

NOTE 2 If the input to the MAC function contains a whole number of bytes, then the function specified here is identical to that described in RFC 4418[6].

6.2.2 Requirements

Before the use of UMAC, the following parameters shall be agreed upon:

— A block cipher standardized in ISO/IEC 18033-3. The choice of a block cipher determines the key length $|K|$ and the block length $|B|$.

— A tag length, *taglen*, which shall be either 4, 8, 12 or 16 octets.

— The length of the nonce, which shall be between 8 and $|B|$ bits.

6.2.3 Notation and auxiliary functions

6.2.3.1 Operations on strings

In contrast to the remainder of this part of ISO/IEC 9797, the specification of UMAC uses a starting index of 1 when numbering elements in a sequence. Thus, for UMAC, $S[i]$ denotes the ith octet of the string S, where $i \geq 1$.

6.2.3.2 Auxiliary function KDF

This key-derivation function generates pseudorandom bits. It returns *numoctets* output octets.

INPUT: Master key K, (*keylen*)-octet string
index, a non-negative integer less than 2^{64}
numoctets, a non-negative integer less than 2^{64}

OUTPUT: Y, (*numoctets*)-octet string

a) n = ceil($numoctets$ / $blocklen$)

b) Set Y to the empty string

c) for i = 1 to n do

   1) T = uint2bitstr($index$, $blocklen$-8) || uint2bitstr(i, 8)

   2) T = Enc(K, T)

   3) Y = Y || T

d) Y = Y[1...$numoctets$]

e) Output Y

NOTE The key-derivation function KDF uses a block cipher in counter mode as defined in ISO/IEC 10116[8].

6.2.3.3 Auxiliary function PDF

This pad-derivation function takes a key and a nonce and returns a pseudorandom padding sequence for use in tag generation. A pad of length 4, 8, 12, or 16 octets can be generated.

INPUT: Master key K, ($keylen$)-octet string
Nonce N, string of length 1 to $blocklen$ octets
Tag length $taglen$, the integer 4, 8, 12 or 16

OUTPUT: Y, ($taglen$)-octets string

a) PDFnonce = N

b) if ($taglen$ = 4 or $taglen$ = 8)

   1) $index$ = bitstr2uint(N) mod ($blocklen$/$taglen$)

   2) $PDFnonce$ = $N \oplus$ uint2bitstr($index$, octetlength(N))

c) padlen = blocklen - octetlength(PDFnonce)

d) PDFnonce = PDFnonce || $0^{padlen*8}$

e) K' = KDF(K, 0, $keylen$)

f) T = Enc(K', PDFnonce)

g) if ($taglen$ = 4 or $taglen$ = 8)

   1) Y = T[(index * taglen) + 1 ... (index * taglen) + taglen]

h) else

   1) Y = T[1 ... $taglen$]

i) Output Y

NOTE Padding sequences generated using nonces that differ only in their last bit (when generating 8-octet pads) or last two bits (when generating 4-octet pads) are derived from the same block cipher encryption. This allows caching and sharing a single block cipher invocation for sequential nonces.

6.2.3.4 Auxiliary function NH

NH ("Non-linear Hash-function") is a universal hash-function.

NOTE The NH universal hash-function was introduced by Black et al.[2].

INPUT: *Key*, 1024-octet string
Msg, string of octets, whose octet length is an integer multiple of 32 and less than or equal to 1024

OUTPUT: *Y*, 8-octet string

Break *Msg* and *Key* into 4-octet blocks:

a) t = octetlength(Msg) / 4

b) Divide *Msg* into 4-octet strings $M_1, M_2, ..., M_t$, so that $Msg = M_1 \| M_2 \| ... \| M_t$.

c) Let $K_1, K_2, ..., K_t$ be 4-octet strings so that $K_1 \| K_2 \| ... \| K_t$ is a prefix of *Key* (the leftmost 4t octets of *Key*).

d) $Y = 0^{64}$

e) $i = 1$

f) while ($i < t$) do

   1) $Y = Y +_{64} ((M_{i+0} +_{32} K_{i+0}) *_{64} (M_{i+4} +_{32} K_{i+4}))$

   2) $Y = Y +_{64} ((M_{i+1} +_{32} K_{i+1}) *_{64} (M_{i+5} +_{32} K_{i+5}))$

   3) $Y = Y +_{64} ((M_{i+2} +_{32} K_{i+2}) *_{64} (M_{i+6} +_{32} K_{i+6}))$

   4) $Y = Y +_{64} ((M_{i+3} +_{32} K_{i+3}) *_{64} (M_{i+7} +_{32} K_{i+7}))$

   5) $i = i + 8$

g) Return Y

NOTE This routine is applied directly to every bit of input data, and therefore optimized implementation of it yields great benefit. It can be performed on the 4-octet blocks, pairing words for multiplication which are 4 apart to accommodate vector-parallelism.

6.2.3.5 Auxiliary function ENDIAN-SWAP

The function ENDIAN-SWAP converts a string of 4-octet words from little-endian to big-endian, or vice versa.

INPUT: *S*, string with length divisible by 4 octets

OUTPUT: *T*, string *S* with each 4-octet word endian-reversed

a) n = octetlength(S) / 4

b) Let $S_1, S_2, ..., S_n$ be strings of length 4 octets so that $S_1 \| S_2 \| ... \| S_n = S$.

c) Set T to the empty string

d) for $i = 1$ to n do

   1) Let W_1, W_2, W_3, W_4 be octets so that $W_1 \| W_2 \| W_3 \| W_4 = S_i$

   2) $S_{Reversed} = \mathrm{W}_4 \| \mathrm{W}_3 \| \mathrm{W}_2 \| \mathrm{W}_1$

   3) $\mathrm{T} = \mathrm{T} \| S_{Reversed}$

e) Output T

6.2.3.6 Auxiliary hash-function POLY

The function POLY is a polynomial hash-function used in the second layer hash-function L2-HASH, see 6.2.7.2.

INPUT: *wordbits*, the integer 64 or 128
maxwordrange, positive integer less than $2^{wordbits}$
key, integer in the range 0 ... prime(*wordbits*) - 1
Msg, string with length divisible by (*wordbits* / 8) octets

OUTPUT: *y*, integer in the range 0 ... prime(*wordbits*) - 1

a) *wordoctets* = *wordbits* / 8

b) p = prime(*wordbits*)

c) *offset* = $2^{wordbits} - p$

d) *marker* = $p - 1$

e) n = octetlength(*Msg*) / *wordoctets*

f) Let $M_1, M_2, ..., M_n$ be strings of length *wordoctets* octets so that $Msg = M_1 \| M_2 \| ... \| M_n$

g) $y = 1$

h) for $i = 1$ to n do

  1) m = bitstr2uint(M_i)

  2) if ($m \geq maxwordrange$) then

    i) $y = (key * y + marker) \bmod p$

    ii) $y = (key * y + (m - offset)) \bmod p$

  3) else

    i) $y = (key * y + m) \bmod p$

i) Output y

6.2.4 Key preprocessing

UMAC uses a block cipher Enc. The block cipher shall be chosen such that *blocklen* is at least 16 and is a power of two.

NOTE 1 It is recommended to use AES-128 for UMAC. In this case, we have blocklen = 16 and keylen = 16.

NOTE 2 If several messages have to be authenticated, it makes sense to buffer the hash key K_H, since it can be re-used. Only the encryption key K_E has to be re-computed for each new message.

INPUT: Master key K, (*keylen*)-octet string
Nonce N, string of length 1 to *blocklen* octets
Tag length *taglen*, the integer 4, 8, 12 or 16

OUTPUT: Hash key K_H = (*L1Key, L2Key, L3Key1, L3Key2*), string of variable length
Encryption key K_E, (*taglen*)-octet string

a) *iters* = *taglen* / 4

b) *L1Key* = KDF(K, 1, 1024 + (*iters* - 1) * 16)

c) *L2Key* = KDF(*K*, 2, *iters* * 24)

d) *L3Key1* = KDF(*K*, 3, *iters* * 64)

e) *L3Key2* = KDF(*K*, 4, *iters* * 4)

f) K_E = PDF(*K, N, taglen*)

g) Output K_H = (*L1Key, L2Key, L3Key1, L3Key2*), K_E

6.2.5 Message preprocessing

Messages to be hashed are viewed as strings of bits that are zero-padded on the right to an appropriate octet length. Once the message is padded, all strings are viewed as strings of octets.

NOTE Message data is read little-endian to speed tag generation on little-endian computers.

6.2.6 Message hashing

INPUT: Hash key K_H = (*L1Key, L2Key, L3Key1, L3Key2*), string of variable length
Encryption key K_E, (*taglen*)-octet string
Message *M*, string of length less than 2^{67} octets
taglen, the integer 4, 8, 12 or 16

OUTPUT: Tag *H*, (*taglen*)-octet string

a) Set *H* to the empty string

b) for *i* = 1 to (*taglen* / 4) do

   1) $L1Key_i = L1Key$ [(*i*-1) * 16 + 1 ... (*i*-1) * 16 + 1024]

   2) $L2Key_i = L2Key$ [(*i*-1) * 24 + 1 ... *i* * 24]

   3) $L3Key1_i = L3Key1$ [(*i*-1) * 64 + 1 ... *i* * 64]

   4) $L3Key2_i = L3Key2$ [(*i*-1) * 4 + 1 ... *i* * 4]

   5) *A* = L1-HASH($L1Key_i$, *M*)

   6) if (bitlength(*M*) <= bitlength($L1Key_i$)) then

      i) $B = 0^{64} \,||\, A$

   7) else

      i) *B* = L2-HASH($L2Key_i$, *A*)

   8) *C* = L3-HASH($\mathrm{L3Key1_i}$, $\mathrm{L3Key2_i}$, B)

   9) $H = H \,||\, C$

c) Output *H*

6.2.7 Layered hash-functions

6.2.7.1 First-layer hash-function L1-HASH

The first-layer hash breaks the message into 1024-octet blocks (padding if necessary the final block) and then endian-adjusts and hashes each with the function NH. Concatenating the results forms a string, which is up to 128 times shorter than the original.

INPUT: *L1Key*, 1024-octet string
L1Msg, string of length less than 2^{67} octets

OUTPUT: *H1*, string of length (8 * ceil(bitlength(*L1Msg*)/8192)) octets

a) t = max(ceil(bitlength(*L1Msg*)/8192), 1)

b) Divide *L1Msg* into strings $M_1, M_2, ..., M_t$, so that *L1Msg* = M_1 || M_2 || ... || M_t, and octetlength(M_i) = 1024 for all $1 \le i \le t$-1.

c) *Len* = uint2bitstr(1024 * 8, 8)

d) *H1* = <empty string>

e) for i = 1 to t-1 do

   1) ENDIAN-SWAP(M_i)

   2) *H1* = *H1* || (NH(*L1Key*, M_i) $+_{64}$ *Len*)

f) *Len* = uint2bitstr(bitlength(M_t), 8)

g) M_t = zeropad(M_t, 32)

h) ENDIAN-SWAP(M_t)

i) *H1* = *H1* || (NH(*L1Key*, M_t) $+_{64}$ *Len*)

j) Output *H1*

6.2.7.2 Second-layer hash-function L2-HASH

The second-layer rehashes the L1-HASH output using a polynomial hash called POLY. If the L1-HASH output is long, then POLY is called once on a prefix of the L1-HASH output and called using different settings on the remainder. This two-step hashing of the L1-HASH output is needed only if the message length is greater than 16 megaoctets.

NOTE Careful implementation of POLY is necessary to avoid a possible timing attack (see [1] for more information).

INPUT: *L2Key*, 24-octet string
L2Msg, string of length less than 2^{64} octets

OUTPUT: *H2*, 16-octet string

a) *Mask64* = uint2bitstr(0x 01FFFFFF 01FFFFFF, 8)

b) *Mask128* = uint2bitstr(0x 01FFFFFF 01FFFFFF 01FFFFFF 01FFFFFF, 16)

c) *k64* = bitstr2uint(*L2Key*[1 ... 8] $\wedge$ *Mask64*)

d) *k128* = bitstr2uint(*L2Key*[9 ... 24] $\wedge$ *Mask128*)

e) if (octetlength(*L2Msg*) $\le 2^{17}$) then

   1) y = POLY(64, $2^{64} - 2^{32}$, *k*64, *L2Msg*)

f) else

   1) M_1 = *L2Msg*[1 ... 2^{17}]

   2) M_2 = *L2Msg*[2^{17} + 1 ... octetlength(*L2Msg*)]

   3) M_2 = zeropad(M_2 || uint2bitstr(0x80,1), 16)

4) y = POLY(64, $2^{64} - 2^{32}$, *k64*, M_1)

5) y = POLY(128, $2^{128} - 2^{96}$, *k128*, uint2bitstr(y, 16) || M_2)

g) *H2* = uint2bitstr(y, 16)

h) Return *H2*

6.2.7.3 Third-layer hash-function L3-HASH

The output from L2-HASH is 16 octets long. This final hash-function hashes the 16-octet string to a fixed length of 4 octets.

INPUT: *K1*, 64-octet string
K2, 4-octet string
Msg, 16-octet string

OUTPUT: *H3*, 4-octet string

a) $y = 0$

b) Break *Msg* and *K1* into 8 blocks and convert to integers:

1) for i = 1 to 8 do

i) M_i = *Msg* [(i - 1) * 2 +1 ... i * 2]

ii) K_i = *K1*[(i - 1) * 8 +1 ... i * 8]

iii) m_i = bitstr2uint(M_i)

iv) k_i = bitstr2uint(K_i) mod prime(36)

c) Inner-product hash, extract last 32 bits and affine-translate:

1) $y = (m_1 * k_1 + ... + m_8 * k_8)$ mod prime(36)

2) $y = y \bmod 2^{32}$

3) *H3* = uint2bitstr(y, 4)

4) $H3 = H3 \oplus K2$

d) Output *H3*

6.2.8 Finalization

INPUT: Encryption key K_E, string of *taglen* octets
Hash value H, string of *taglen* octets

OUTPUT: MAC of length *taglen* octets

a) MAC = $K_E \oplus H$

b) Output MAC

6.3 Badger

6.3.1 Description of Badger

Badger is a MAC algorithm that uses a 128-bit key K and a 64-bit nonce N. It processes a message of length up to 2^{61} –1 octets into an authentication tag of length *taglen*, which can be 4, 8, 12, 16 or 20 octets. The input message comprises a whole number of octets (i.e. the bit-length of the message shall be a multiple of 8). Badger uses a stream cipher (see ISO/IEC 18033-4) or pseudorandom generator (see ISO/IEC 18031) PRG.

NOTE Badger was proposed in [3] and is based on the classical hash tree construction by Wegman and Carter[10].

6.3.2 Requirements

Before the use of Badger, the following parameters shall be agreed upon:

— A stream cipher from ISO/IEC 18033-4 or a pseudorandom generator that complies with ISO/IEC 18031.

— A tag length, *taglen*, which shall be 4, 8, 12, 16 or 20 octets.

6.3.3 Notation and auxiliary functions

6.3.3.1 Auxiliary function ENH

ENH ("Enhanced Non-linear Hash-function") is a universal hash-function.

NOTE The ENH universal hash-function was introduced by Boesgaard et al.[3]. It is based on the NH function of Black et al.[2] that is used in UMAC.

INPUT: Key *LKey*, 8-octet string
Message *Left*, 8-octet string
Message *Right*, 8-octet string

OUTPUT: Hash value *LHash*, 8-octet string

a) k_L = octetstr2uint(*LKey*[0..3]), k_U = octetstr2uint(*LKey*[4...7])

b) $m1_L$ = octetstr2uint(*Right*[0...3]), $m1_U$ = octetstr2uint(*Right*[4...7])

c) $m2$ = octetstr2uint(*Left*[0...7])

d) $h_L = (m1_L + k_L) \bmod 2^{32}$

e) $h_U = (m1_U + k_U) \bmod 2^{32}$

f) Let $h' = ((h_U * h_L) + m2) \bmod 2^{64}$

g) *LHash* = uint2octetstr(h',8)

h) Output *LHash*

6.3.4 Key preprocessing

The key length for Badger is 16 octets, and the nonce length is 8 octets. If the generator requires longer keys or nonces, the remaining octets can be padded with zeroes. The nonce value shall be different from the all-one vector. The PRG is assumed to have the following interfaces:

— PRG_Init(K,N) initializes the inner state of the PRG with key K and nonce N.

— PRG_Next(n) produces the next n output bits from the PRG.

Using these functions, hash and encryption keys are computed as follows.

NOTE If several messages have to be authenticated, it makes sense to buffer the hash key K_H, since it can be re-used. Only the encryption key K_E has to be re-computed for each new message.

INPUT: Master key K, 16-octet string
Nonce N, 8-octet string
Bit length *maxlen* of the longest possible input message, integer multiple of 8 with $0 \leq maxlen \leq$
$2^{64} - 8$
Tag length *taglen*, integer (one of 4,8,12,16,20)

OUTPUT: Hash key $K_H = (KL, kf)$, string of variable length
(where KL is a vector of 8-octet strings and kf is a vector of 4-octet integers)
Encryption key K_E, (*taglen*)-octet string

a) PRG_Init(K, 1^{64})

b) words_used = 0

c) $u = taglen / 4$

d) $v = \max\{1, \text{ceil}(\log_2(maxlen)) - 6\}$

e) for j = 1 to 6:

   1) for i = 1 to u:

      i) $kf_{j,i}$ = octetstr2uint(PRG_Next(32))

      ii) *words_used* = *words_used* + 1

f) for j = 1 to 6:

   1) for i = 1 to u:

      i) while($k_{j,i} \geq$ prime(32))

         I) $kf_{j,i}$ = octetstr2uint(PRG_Next(32))

         II) *words_used* = *words_used* + 1

g) while(*words_used* mod 4 ≠ 0):

   1) discard PRG_Next(32)

   2) *words_used* = *words_used* + 1

h) for j = 1 to v:

   1) for i = 1 to u:

      i) $KL_{j,i}$ = PRG_Next(64)

i) PRG_Init(K, N)

j) K_E = PRG_Next(32*u)

k) Output $K_H = (KL, kf)$, K_E

6.3.5 Message preprocessing

No message preprocessing is necessary for Badger.

6.3.6 Message hashing

The message is hashed by computing the following polynomial expression:

INPUT: Hash key $K_H = (KL, kf)$, string of variable length
Message M, string of length at most 2^{61} -1 octets
Tag length *taglen*, integer (one of 4,8,12,16,20)

OUTPUT: Hash value H, (*taglen*)-octet string

a) len = bitlength(M) as 64-bit integer

b) if $len = 0$:

   1) $M_1 = \ldots = M_u = 0^{64}$

c) else:

   1) if $len \bmod 64 \neq 0$:

      i) Append zero bits in the most significant bits until length len of M is a multiple of 64 bits.

   2) for i = 1 to u:

      i) $M_i = M$

      ii) $v' = \max\{1, \mathrm{ceil}(\log_2(len)) - 6\}$

      iii) for j = 1 to v':

         I) t = octetlength(M_i) / 8

         II) Divide M_i into 8-octet blocks $B_1,\ldots,B_t$ such that $M_i = B_t \mid\mid \ldots \mid\mid B_1$

         III) *if* t is even:

            a) $M_i = \mathrm{ENH}(KL_{j,i}, B_t, B_{t-1}) \mid\mid \ldots \mid\mid \mathrm{ENH}(KL_{j,i}, B_2, B_1)$

         IV) else:

            a) $M_i = \mathrm{B}_t \mid\mid \mathrm{ENH}(KL_{j,i}, B_{t-1}, B_{t-2}) \mid\mid \ldots \mid\mid \mathrm{ENH}(KL_{j,i}, B_2, B_1)$

d) for i = 1 to u:

   1) $Q_i = 0^7 \mid\mid len \mid\mid M_i$

   2) Divide Q_i into 27-bit blocks $B_1,\ldots,B_5$ such that $Q_i = B_5 \mid\mid \ldots \mid\mid B_1$.

   3) Pad each block $B_1,\ldots,B_5$ with zeroes in the most significant bits such that it is 4 octets long.

   4) for j = 1 to 5:

      i) b_j = octetstr2uint(B_j)

   5) $s_i = ((b_1 * kf_{1,i}) + \ldots + (b_5 * kf_{5,i}) + kf_{6,i}) \bmod \mathrm{prime}(32)$

   6) S_i = uint2octetstr(s_i,4)

e) $H = S_u \mid\mid \ldots \mid\mid S_1$

f) Output H

6.3.7 Finalization

INPUT: Encryption key K_E, (*taglen*)-octet string
Hash value H, (*taglen*)-octet string

OUTPUT: Message authentication code MAC, (*taglen*)-octet string

a) MAC $= K_E \oplus H$

b) Output MAC

6.4 Poly1305-AES

6.4.1 Description of Poly1305-AES

Poly1305 is a MAC algorithm that uses a 256-bit key K (with 22 bits set to zero) and a 128-bit nonce N. It accepts messages of arbitrary octet length l and produces a 128-bit MAC. The input message comprises a whole number of octets, i.e. the bit-length of the message shall be a multiple of 8.

NOTE Poly1305 was proposed in [1] and is based on polynomial hashing. Compared to a naive implementation, the performance of Poly1305-AES can be improved significantly by following the implementation advice given in [1].

6.4.2 Requirements

The use of Poly1305-AES requires no additional parameters to be agreed upon.

6.4.3 Key preprocessing

The 32-octet master key K has a special formatting, in that certain bits have to be zero. These bits are:

— The 4 most significant bits of $K[3]$, $K[7]$, $K[11]$, $K[15]$.

— The 2 least significant bits of $K[4]$, $K[8]$, $K[12]$.

The master key is then simply split into a hash and an encryption key, as follows:

INPUT: Master key K, 32-octet string

OUTPUT: Hash key K_H , 16-octet string
Encryption key K_E, 16-octet string

a) $K_H = K[0 \ldots 15]$

b) $K_E = K[16 \ldots 31]$

c) Output K_H, K_E

6.4.4 Message preprocessing

The message is preprocessed as follows:

INPUT: Message M, l_o-octet string

OUTPUT: number of message blocks s, integer
preprocessed message $c_1,\ldots,c_s$, sequence of 17-octet integers

a) Let l_o = octetlength(M)

b) Let s = ceil(l_o/16)

c) Let t = floor(l_o/16)

d) For $i = 0,\ldots,t\text{-}1$:

   1) c_{i+1} = octetstr2uint($M[16i \ldots 16i+15]$) + 2^{128}

e) If $s > t$:

   1) Let $r = l_o \bmod 16$

2) c_s = octetstr2uint ($M[16t \ldots l_o\text{-}1]$) + 2^{8r}

f) Output s; $c_1,\ldots,c_s$

6.4.5 Message hashing

The message is hashed by computing the following polynomial expression:

INPUT: Hash key K_H, 16-octet string
number of message blocks s, integer
preprocessed message $c_1,\ldots,c_s$, sequence of 17-octet integers

OUTPUT: Hash value H, 16-octet integer

a) r = octetstr2uint(K_H)

b) $H' = (c_1 * r^s + c_2 * r^{s-1} + \ldots + c_s * r^1)$ mod prime(130)

c) Let $H = H'$ mod 2^{128}

d) Output H

6.4.6 Finalization

Finally, the message is encrypted using the AES-128 block cipher, which is described in ISO/IEC 18033-3.

INPUT: Hash value H, 16-octet integer
Encryption key K_E, 16-octet string
Nonce N, 16-octet string

OUTPUT: Message authentication code MAC, 16-octet string

a) Let S = AES-128(K_E, N)

b) s = octetstr2uint(S)

c) Let $mac = (H + s)$ mod 2^{128}

d) MAC = uint2octetstr(mac,16)

e) Output MAC

6.5 GMAC

6.5.1 Description of GMAC

[A1⟩ GMAC can be used with any block cipher from ISO/IEC 18033-3 that has a block length of 128 bits. The resulting MAC is t bits long, where t is a multiple of 8 satisfying $96 \leq t \leq 128$ ($t = 64$ is also permitted for specialized applications – see 6.5.2).⟨A1] The resulting MAC is t bits long, where t is a multiple of 8 and satisfies $64 \leq t \leq 128$. The length of the input message shall be less than or equal to 2^{64} blocks.

NOTE This mechanism is a special case of the GCM (for Galois/Counter Mode) specified in ISO/IEC 19772 where no data is encrypted. GCM is due to McGrew and Viega [9].

6.5.2 Requirements

Before the use of GMAC, the following parameters shall be agreed upon:

— A block cipher from ISO/IEC 18033-3 with a block length of 128 bits. The choice of a block cipher determines the key length $|K|$.

— ⟦A1⟩ The tag length, t, shall be selected such that t is a multiple of 8 satisfying $96 \leq t \leq 128$. The only permitted exception to this is tag length $t = 64$. However, this tag length is only permitted for specialized applications, and should only be used with great care.

NOTE For some voice or video applications, short authentication tags (i.e. where $t = 64$) can be appropriate. In such applications the forgery of some fraction of individual authenticated "packets" can be tolerable, because each packet of data in a large stream can carry very little of the overall meaning. However, even for such applications, short tags can be problematic for GMAC as a result of targeted forgery attacks of the type documented in Appendix B of [9]. Detailed guidance on use of tag length $t = 64$ is provided in Appendix C of [9]. ⟦A1⟧

— The length of the nonce.

6.5.3 Notation and auxiliary functions

6.5.3.1 Auxiliary function GHASH

The function GHASH takes as input a 128-bit block H and two arbitrary length strings of bits W and Z, and gives a 128-bit block as output.

INPUT: 128-bit block H
arbitrary length strings of bits W and Z

OUTPUT: 128-bit value X_{k+l+1}

a) Let k and u be the unique integers such that bitlength(W) = $128(k\text{-}1)+u$ and $0 < u \leq 128$. Let $W_1, W_2, \ldots, W_k$ be the sequence of 128-bit blocks (with the possible exception of W_k which contains the final u bits of W) obtained by partitioning W.

b) Let l and v be the unique integers such that bitlength(Z) = $128(l\text{-}1)+v$ and $0 < v \leq 128$. Let $Z_1, Z_2, \ldots, Z_l$ be the sequence of 128-bit blocks (with the possible exception of Z_l, which contains the final v bits of Z) obtained by partitioning Z.

c) Compute 128-bit value X_{k+l+1} using the following recursion:

1) $X_0 = 0^{128}$.

2) $X_i = (X_{i\text{-}1} \oplus W_i) \bullet H,$ $1 \leq i \leq k\text{-}1$ (this step is omitted if $k \leq 1$).

3) $X_k = (X_{k\text{-}1} \oplus (W_k || 0^{128\text{-}u})) \bullet H$ (this step is omitted if k=0).

4) $X_i = (X_{i\text{-}1} \oplus Z_{i\text{-}k}) \bullet H,$ $k+1 \leq i \leq k+l\text{-}1$ (this step is omitted if $l \leq 1$).

5) $X_{k+l} = (X_{k+l\text{-}1} \oplus (Z_l || 0^{128\text{-}v})) \bullet H$ (this step is omitted if l =0).

6) $X_{k+l+1} = (X_{k+l} \oplus \text{uint2bitstr}(\text{bitlength}(W),8) \,||\, \text{uint2bitstr}(\text{bitlength}(Z),8)) \bullet H$.

d) Output X_{k+l+1}.

6.5.4 Key preprocessing

The master key K is used to derive the hash key K_H and the encryption key K_E as follows

INPUT: Master key K

OUTPUT: Hash key K_H
Encryption key K_E

a) $K_H = \text{Enc}(K, 0^{128})$

b) $K_E = K$

c) Output K_H, K_E

6.5.5 Message preprocessing

No message preprocessing is necessary for GMAC.

6.5.6 Message hashing

INPUT: Message M
Hash key K_H

OUTPUT: Hash value H

a) H = GHASH$(K_H, M, \{\})$

b) Output H

6.5.7 Finalization

A variable length nonce N shall be selected. This value shall be distinct for every message to be protected, and shall be made available to the recipient of the message. However, it is not necessary that this value be unpredictable or secret.

NOTE The value N could, for example, be generated using a counter maintained by the originator, and sent in clear text along with the protected message.

INPUT: Hash value H
Encryption key K_E
Nonce N

OUTPUT: Message authentication code MAC, t-bit string

a) If bitlength(N) = 96 then let $Y_0 = N \,||\, 0^{31} \,||\, 1$. Otherwise let Y_0 = GHASH(K_H, {}, N).

b) Let MAC = $(H \oplus \mathrm{Enc}(K_E, Y_0))|_t$.

c) Output MAC.

Annex A
(normative)

Object Identifiers

This annex lists the object identifiers assigned to algorithms specified in this part of ISO/IEC 9797.

```
MessageAuthenticationCodesPart3 {

     iso(1) standard(0) message-authentication-codes(9797) part3(3)

          asn1-module(0) algorithm-object-identifiers(0)

}

DEFINITIONS EXPLICIT TAGS ::= BEGIN

EXPORTS ALL;

IMPORTS

    BlockAlgorithms

         FROM EncryptionAlgorithms-3 { iso(1) standard(0)

              encryption-algorithms(18033) part(3)

              asn1-module(0) algorithm-object-identifiers(0) }

    StreamCipherAlgorighms

         FROM EncryptionAlgorithms-4 { iso(1) standard(0)

              encryption-algorithms(18033) part(4)

              asn1-module(0) algorithm-object-identifiers(0) };

OID ::= OBJECT IDENTIFIER -- Alias

-- Synonym

is9797-3 OID ::= {iso standard message-authentication-codes(9797) part3(3)}

-- OID assignments

id-umac OID ::= {is9797-3 umac(1)}

id-badger OID ::= {is9797-3 badger(2)}

id-poly1305 OID ::= {is9797-3 poly1305-aes(3)}

id-gmac OID ::= {is9797-3 gmac(4)}


-- mac algorithm identifier type and the set of recognized mac algorithms

MessageAuthenticationCode ::= AlgorithmIdentifier {{ MacAlgorithms }}

MacAlgorithms ALGORITHM ::= {

     { OID id-umac PARMS UmacParameters } |
```

```
     { OID id-badger PARMS BadgerParameters } |
     { OID id-poly1305 PARMS NullParameters } |
     { OID id-gmac PARMS GmacParameters },
     ... -- expect additional algorithms --
}
-- mac parameter types definitions
UmacParameters ::= SEQUENCE {
    bcAlgo BlockAlgorithms,
    taglength INTEGER,
    noncelength INTEGER
}
BadgerParameters ::= SEQUENCE {
    scAlgo StreamCipherAlgorithms,
    taglength INTEGER,
}
GmacParameters ::= SEQUENCE {
    bcAlgo BlockAlgorithms,
    taglength INTEGER,
    noncelength INTEGER
}
END -- MessageAuthenticationCodesPart3 --
```

Annex B
(informative)

[A1⟩ Numerical Examples ⟨A1]

B.1 UMAC

This clause contains some UMAC [A1⟩ numerical examples ⟨A1], using AES-128 as the block cipher. Table B.1 lists the tags generated by UMAC using the 16-byte key *K* and 8-byte nonce *N*.

— *K* = "abcdefghijklmnop"

— *N* = "bcdefghi"

Table B.1 — UMAC [A1⟩ numerical examples ⟨A1]

Message	UMAC-32	UMAC-64	UMAC-96	UMAC-128
<empty>	113145FB	6E155FAD-26900BE1	32FEDB100C79AD-58F07FF764	32FEDB100C79AD-58F07FF7643CC60465
'a' * 3	3B91D102	44B5CB542F220104	185E4FE905CBA7BD-85E4C2DC	185E4FE905CBA7BD85E4C2DC3D-117D8D
'a' * 2^{10}	599B350B	26BF2F5D60118BD9	7A54ABE04AF82D60F-B298C3C	7A54ABE04AF82D60FB298C3CBD-195BCB
'a' * 2^{15}	58DCF532	27F8EF643B-0D118D	7B136BD911E4B734286E-F2BE	7B136BD911E4B734286EF2BE-501F2C3C

B.2 Badger

This clause contains some Badger [A1⟩ numerical examples ⟨A1], using Rabbit as the stream cipher. Table B.2 lists the tags generated by Badger using the following key *K* and *IV*:

— *K* = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

— *IV* = 00 01 02 03 04 05 06 07

Table B.2 — Badger [A1⟩ numerical examples ⟨A1]

Message	Badger tag
<empty>	54 6D 3A 85 F8 CB FA D9 E0 58 50 58 2C AC 3D E4
00	5F AA AB 85 AC BE 04 48 1D D6 34 D0 FA D9 FA FA
01	47 EA 18 A1 99 AE 07 31 7C A5 AC C9 37 2F 55 85
00 01 02 03 04 05 06 07 08	F7 02 3D 65 CF 66 69 23 47 A0 8B 5F 93 55 84 27

B.3 Poly1305-AES

This clause contains some [A1⟩ numerical examples ⟨A1] for Poly1305-AES in Table B.3.

Table B.3 — Poly1305-AES A1⟩ numerical examples ⟨A1

Numerical example #1	**Message**	<empty>
	Encryption key K_E	75 de aa 25 c0 9f 20 8e 1d c4 ce 6b 5c ad 3f bf
	Hash key K_H	a0 f3 08 00 00 f4 64 00 d0 c7 e9 07 6c 83 44 03
	Nonce N	61 ee 09 21 8d 29 b0 aa ed 7e 15 4a 2c 55 09 cc
	Poly1305-AES tag	dd 3f ab 22 51 f1 1a c7 59 f0 88 71 29 cc 2e e7
Numerical example #2	**Message**	f3 f6
	Encryption key K_E	ec 07 4c 83 55 80 74 17 01 42 5b 62 32 35 ad d6
	Hash key K_H	85 1f c4 0c 34 67 ac 0b e0 5c c2 04 04 f3 f7 00
	Nonce N	fb 44 73 50 c4 e8 68 c5 2a c3 27 5c f9 d4 32 7e
	Poly1305-AES tag	f4 c6 33 c3 04 4f c1 45 f8 4f 33 5c b8 19 53 de
Numerical example #3	**Message**	66 3c ea 19 0f fb 83 d8 95 93 f3 f4 76 b6 bc 24 d7 e6 79 10 7e a2 6a db 8c af 66 52 d0 65 61 36
	Encryption key K_E	6a cb 5f 61 a7 17 6d d3 20 c5 c1 eb 2e dc dc 74
	Hash key K_H	48 44 3d 0b b0 d2 11 09 c8 9a 10 0b 5c e2 c2 08
	Nonce N	ae 21 2a 55 39 97 29 59 5d ea 45 8b c6 21 ff 0e
	Poly1305-AES tag	0e e1 c1 6b b7 3f 0f 4f d1 98 81 75 3c 01 cd be
Numerical example #4	**Message**	ab 08 12 72 4a 7f 1e 34 27 42 cb ed 37 4d 94 d1 36 c6 b8 79 5d 45 b3 81 98 30 f2 c0 44 91 fa f0 99 0c 62 e4 8b 80 18 b2 c3 e4 a0 fa 31 34 cb 67 fa 83 e1 58 c9 94 d9 61 c4 cb 21 09 5c 1b f9
	Encryption key K_E	e1 a5 66 8a 4d 5b 66 a5 f6 8c c5 42 4e d5 98 2d
	Hash key K_H	12 97 6a 08 c4 42 6d 0c e8 a8 24 07 c4 f4 82 07
	Nonce N	9a e8 31 e7 43 97 8d 3a 23 52 7c 71 28 14 9e 3a
	Poly1305-AES tag	51 54 ad 0d 2c b2 6e 01 27 4f c5 11 48 49 1f 1b

B.4 GMAC

This clause contains some A1⟩ numerical examples ⟨A1 for GMAC using AES-128 as the block cipher in Table B.4.

Table B.4 — GMAC A1⟩ numerical examples ⟨A1

Numerical example #1	**Message**	<empty>
	Key K	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
	Nonce N	00 00 00 00 00 00 00 00 00 00 00 00
	GMAC tag	58 e2 fc ce fa 7e 30 61 36 7f 1d 57 a4 e7 45 5a
Numerical example #2	**Message**	fe ed fa ce de ad be ef fe ed fa ce de ad be ef
	Key K	fe ff e9 92 86 65 73 1c 6d 6a 8f 94 67 30 83 08
	Nonce N	ca fe ba be fa ce db ad de ca f8 88
	GMAC tag	54 df 47 4f 4e 71 a9 ef 8a 09 bf 30 da 7b 1a 92

Numerical example #3	**Message**	fe ed fa ce de ad be ef fe ed fa ce de ad be ef ab ad da d2 42 83 1e c2 21 77 74 24 4b 72 21 b7
	Key *K*	fe ff e9 92 86 65 73 1c 6d 6a 8f 94 67 30 83 08
	Nonce *N*	ca fe ba be fa ce db ad de ca f8 88
	GMAC tag	1c be 39 36 e5 53 b0 8f 25 c0 8d 7b 8d c3 9f db

Annex C
(informative)

Security Information

Several attacks against universal hash-function based MAC algorithms are described in [5]. In contrast to other types of MAC algorithms, a small number of forgeries can lead to key recovery for MAC functions based on a universal hash-function, and thus to a complete collapse of security

To defend against these attacks, it is highly recommended to implement one or more of the following countermeasures:

a) Increase the security level (e.g. taglen) so that the first forgery is practically impossible.

b) Regularly refresh the complete key used for the MAC algorithm. If possible, one should even refresh the key for every message. Note that not just the number of messages processed under a single key should be limited, but also the total number of message blocks processed under the same key.

c) Both the sender and the receiver need to guarantee/check the uniqueness of nonces used with the same MAC key. When a nonce is reused, the security of the algorithm is severely reduced.

d) Receivers should detect a large number of failed MAC verifications as an attack attempt, and deal with this situation appropriately.

Bibliography

[1] Bernstein, D. J., *The Poly1305-AES message-authentication code*. Proceedings of Fast Software Encryption 2005, LNCS 3557, pp. 32-49, Springer-Verlag, 2005

[2] Black, J., Halevi, S., Krawczyk, H., Krovetz, T. and Rogaway, P. *UMAC: Fast and provably secure message authentication*, Advances in Cryptology - CRYPTO '99, LNCS vol. 1666, pp. 216-233, Springer-Verlag, 1999

[3] Boesgaard, M., Christensen, T. and Zenner, E. Badger. *A fast and provably secure MAC*, Proceedings of Applied Cryptography and Network Security, LNCS vol. 3531, pp. 176-191, Springer-Verlag, 2005

[4] Carter, L. and Wegman, M. Universal classes of hash functions, *Journal of Computer and System Sciences*, 18 (1979), pp. 143-154

[5] Handschuh, H. and Preneel, B. *Key-Recovery Attacks on Universal Hash Function based MAC Algorithms*. Advances in Cryptology - CRYPTO '08, LNCS vol. 5157, pp. 144-161, Springer-Verlag, 2008

[6] IETF RFC 4418, *UMAC: Message Authentication Code using Universal Hashing, March 2006*

[7] ISO/IEC 10181-6, *Information technology — Open Systems Interconnection — Security frameworks for open systems: Integrity framework*

[8] ISO/IEC 10116, *Information technology — Security techniques — Modes of operation for an n-bit block cipher*

[9] A1 National Institute of Standards and Technology, *NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. November 2007 A1

[10] Wegman, M. and Carter, L. New hash functions and their use in authentication and set equality, *Journal of Computer and System Sciences*, 22 (1981), pp. 265-279

British Standards Institution (BSI)

BSI is the national body responsible for preparing British Standards and other standards-related publications, information and services.

BSI is incorporated by Royal Charter. British Standards and other standardization products are published by BSI Standards Limited.

About us

We bring together business, industry, government, consumers, innovators and others to shape their combined experience and expertise into standards -based solutions.

The knowledge embodied in our standards has been carefully assembled in a dependable format and refined through our open consultation process. Organizations of all sizes and across all sectors choose standards to help them achieve their goals.

Information on standards

We can provide you with the knowledge that your organization needs to succeed. Find out more about British Standards by visiting our website at bsigroup.com/standards or contacting our Customer Services team or Knowledge Centre.

Buying standards

You can buy and download PDF versions of BSI publications, including British and adopted European and international standards, through our website at bsigroup.com/shop, where hard copies can also be purchased.

If you need international and foreign standards from other Standards Development Organizations, hard copies can be ordered from our Customer Services team.

Copyright in BSI publications

All the content in BSI publications, including British Standards, is the property of and copyrighted by BSI or some person or entity that owns copyright in the information used (such as the international standardization bodies) and has formally licensed such information to BSI for commercial publication and use.

Save for the provisions below, you may not transfer, share or disseminate any portion of the standard to any other person. You may not adapt, distribute, commercially exploit or publicly display the standard or any portion thereof in any manner whatsoever without BSI's prior written consent.

Storing and using standards

Standards purchased in soft copy format:

- A British Standard purchased in soft copy format is licensed to a sole named user for personal or internal company use only.
- The standard may be stored on more than one device provided that it is accessible by the sole named user only and that only one copy is accessed at any one time.
- A single paper copy may be printed for personal or internal company use only.

Standards purchased in hard copy format:

- A British Standard purchased in hard copy format is for personal or internal company use only.
- It may not be further reproduced – in any format – to create an additional copy. This includes scanning of the document.

If you need more than one copy of the document, or if you wish to share the document on an internal network, you can save money by choosing a subscription product (see 'Subscriptions').

Reproducing extracts

For permission to reproduce content from BSI publications contact the BSI Copyright and Licensing team.

Subscriptions

Our range of subscription services are designed to make using standards easier for you. For further information on our subscription products go to bsigroup.com/subscriptions.

With **British Standards Online (BSOL)** you'll have instant access to over 55,000 British and adopted European and international standards from your desktop. It's available 24/7 and is refreshed daily so you'll always be up to date.

You can keep in touch with standards developments and receive substantial discounts on the purchase price of standards, both in single copy and subscription format, by becoming a **BSI Subscribing Member.**

PLUS is an updating service exclusive to BSI Subscribing Members. You will automatically receive the latest hard copy of your standards when they're revised or replaced.

To find out more about becoming a BSI Subscribing Member and the benefits of membership, please visit bsigroup.com/shop.

With a **Multi-User Network Licence (MUNL)** you are able to host standards publications on your intranet. Licences can cover as few or as many users as you wish. With updates supplied as soon as they're available, you can be sure your documentation is current. For further information, email cservices@bsigroup.com.

Revisions

Our British Standards and other publications are updated by amendment or revision.

We continually improve the quality of our products and services to benefit your business. If you find an inaccuracy or ambiguity within a British Standard or other BSI publication please inform the Knowledge Centre.

Useful Contacts

Customer Services
Tel: +44 345 086 9001
Email: cservices@bsigroup.com

Subscriptions
Tel: +44 345 086 9001
Email: subscriptions@bsigroup.com

Knowledge Centre
Tel: +44 20 8996 7004
Email: knowledgecentre@bsigroup.com

Copyright & Licensing
Tel: +44 20 8996 7070
Email: copyright@bsigroup.com

BSI Group Headquarters

389 Chiswick High Road London W4 4AL UK