

---

---

**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*

IECNORM.COM : Click to view the PDF of ISO/IEC 9797-3:2011

IECNORM.COM : Click to view the full PDF of ISO/IEC 9797-3:2011



**COPYRIGHT PROTECTED DOCUMENT**

© ISO/IEC 2011

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office  
Case postale 56 • CH-1211 Geneva 20  
Tel. + 41 22 749 01 11  
Fax + 41 22 749 09 47  
E-mail [copyright@iso.org](mailto:copyright@iso.org)  
Web [www.iso.org](http://www.iso.org)

Published in Switzerland

# Contents

Page

Foreword .....	iv
Introduction.....	v
<b>1 Scope .....</b>	<b>1</b>
<b>2 Normative references .....</b>	<b>1</b>
<b>3 Terms and definitions .....</b>	<b>1</b>
<b>4 Symbols and abbreviated terms .....</b>	<b>2</b>
<b>5 General model.....</b>	<b>4</b>
<b>6 Mechanisms .....</b>	<b>5</b>
6.1 Introduction.....	5
6.2 UMAC .....	5
6.2.1 Description of UMAC.....	5
6.2.2 Requirements.....	5
6.2.3 Notation and auxiliary functions.....	5
6.2.4 Key preprocessing .....	9
6.2.5 Message preprocessing.....	9
6.2.6 Message hashing.....	9
6.2.7 Layered hash-functions .....	10
6.2.8 Finalization .....	12
6.3 Badger .....	12
6.3.1 Description of Badger.....	12
6.3.2 Requirements.....	12
6.3.3 Notation and auxiliary functions.....	13
6.3.4 Key preprocessing .....	13
6.3.5 Message preprocessing.....	14
6.3.6 Message hashing.....	14
6.3.7 Finalization.....	16
6.4 Poly1305-AES .....	16
6.4.1 Description of Poly1305-AES .....	16
6.4.2 Requirements.....	16
6.4.3 Key preprocessing .....	16
6.4.4 Message preprocessing.....	16
6.4.5 Message hashing.....	17
6.4.6 Finalization .....	17
6.5 GMAC .....	18
6.5.1 Description of GMAC .....	18
6.5.2 Requirements.....	18
6.5.3 Notation and auxiliary functions.....	18
6.5.4 Key preprocessing .....	19
6.5.5 Message preprocessing.....	19
6.5.6 Message hashing.....	19
6.5.7 Finalization .....	19
<b>Annex A (normative) Object Identifiers .....</b>	<b>20</b>
<b>Annex B (informative) Test Vectors .....</b>	<b>22</b>
<b>Annex C (informative) Security Information.....</b>	<b>24</b>
<b>Bibliography.....</b>	<b>25</b>

## 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<sup>[7]</sup>.

IECNORM.COM : Click to view the full PDF of ISO/IEC 9797-3:2011

[IECNORM.COM](http://IECNORM.COM) : Click to view the full PDF of ISO/IEC 9797-3:2011

# 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 Universal hash-functions were introduced by Carter and Wegman<sup>[4]</sup>, and their application in MAC algorithms was first described by Wegman and Carter<sup>[10]</sup>.

## 4 Symbols and abbreviated terms

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

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

$\text{bitlength}(S)$  Length of a string  $S$  in bits.

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

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

$\text{blocklen}$  Block length of the underlying block cipher in octets.

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

$\text{Enc}(K, X)$  Encryption of a plaintext block  $X$  under a key  $K$  using a block cipher  $\text{Enc}$ .

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

$H$  Hash value.

$K$  Master key.

$K_E$  Encryption key.

$K_H$  Hash key.

$\text{keylen}$  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  $S[0] + 2^8 * S[1] + 2^{16} * S[2] + \dots + 2^{8n-8} * S[n-1]$ , where  $n = \text{octetlength}(S)$ .
- 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 FFFFFFFFB
36	$2^{36} - 5$	0x 0000000F FFFFFFFB
64	$2^{64} - 59$	0x FFFFFFFF FFFFFFFC5
128	$2^{128} - 159$	0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFF61
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  $\text{bitstr2uint}(S) = x$ .
- uint2octetstr( $x, n$ ) The  $n$ -octet string  $S$  such that  $x = \text{octetstr2uint}(S)$ .
- $X|_s$  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  $X$ .
- $X \gg 1$  Right shift of a block of bits  $X$  by one position: the leftmost bit of  $Y = X \gg 1$  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,  $\text{zeropad}(S, n) = S \parallel T$ , where  $T$  is the shortest string of zero-bits (possibly empty) so that  $S \parallel T$  is non-empty and  $n$  divides  $\text{octetlength}(S \parallel T)$ .

- ⊕ 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  $B$ .
- ^ 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  $B$ .
- +<sub>32</sub> Addition of two 32-bit strings, resulting in a 32-bit string. More formally,  $S +_{32} T = \text{uint2bitstr}(\text{bitstr2uint}(S) + \text{bitstr2uint}(T) \bmod 2^{32}, 4)$ .
- +<sub>64</sub> Addition of two 64-bit strings, resulting in a 64-bit string. More formally,  $S +_{64} T = \text{uint2bitstr}(\text{bitstr2uint}(S) + \text{bitstr2uint}(T) \bmod 2^{64}, 8)$ .
- \* Multiplication operator on integers.
- \*<sub>64</sub> Multiplication of two 64-bit strings, resulting in a 64-bit string. More formally,  $S *_{64} T = \text{uint2bitstr}(\text{bitstr2uint}(S) * \text{bitstr2uint}(T) \bmod 2^{64}, 8)$ .

NOTE The operations +<sub>32</sub>, +<sub>64</sub> and \*<sub>64</sub> 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<sup>*n*</sup> String consisting of  $n$  zero-bits.
- 1<sup>*n*</sup> 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  $i = 1, 2, \dots, 128$ , perform the following two steps:
  - 1) If  $\text{bit}(V, i) = 1$  then let  $W = W \oplus Z$ ;
  - 2) If  $\text{bit}(Z, 128) = 0$  then let  $Z = Z \gg 1$ ; otherwise let  $Z = (Z \gg 1) \oplus (11100001 || 0^{120})$ .

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.

## 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.<sup>[2]</sup>.

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<sup>[6]</sup>.

#### 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  $i$ th 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)  $n = \text{ceil}(\text{numoctets} / \text{blocklen})$
- b) Set *Y* to the empty string
- c) for *i* = 1 to *n* do
  - 1)  $T = \text{uint2bitstr}(\text{index}, \text{blocklen}-8) \parallel \text{uint2bitstr}(i, 8)$
  - 2)  $T = \text{Enc}(K, T)$
  - 3)  $Y = Y \parallel T$
- d)  $Y = Y[1 \dots \text{numoctets}]$
- e) Output *Y*

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

### 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)  $\text{index} = \text{bitstr2uint}(N) \bmod (\text{blocklen}/\text{taglen})$
  - 2)  $\text{PDFnonce} = N \oplus \text{uint2bitstr}(\text{index}, \text{octetlength}(N))$
- c)  $\text{padlen} = \text{blocklen} - \text{octetlength}(\text{PDFnonce})$
- d)  $\text{PDFnonce} = \text{PDFnonce} \parallel 0^{\text{padlen} \times 8}$
- e)  $K' = \text{KDF}(K, 0, \text{keylen})$
- f)  $T = \text{Enc}(K', \text{PDFnonce})$
- g) if (*taglen* = 4 or *taglen* = 8)
  - 1)  $Y = T[(\text{index} \times \text{taglen}) + 1 \dots (\text{index} \times \text{taglen}) + \text{taglen}]$

h) else

$$1) Y = T[1 \dots \text{taglen}]$$

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.<sup>[2]</sup>.

**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)  $t = \text{octetlength}(Msg) / 4$
- b) Divide  $Msg$  into 4-octet strings  $M_1, M_2, \dots, M_t$ , so that  $Msg = M_1 \parallel M_2 \parallel \dots \parallel M_t$ .
- c) Let  $K_1, K_2, \dots, K_t$  be 4-octet strings so that  $K_1 \parallel K_2 \parallel \dots \parallel K_t$  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)  $n = \text{octetlength}(S) / 4$
- b) Let  $S_1, S_2, \dots, S_n$  be strings of length 4 octets so that  $S_1 \parallel S_2 \parallel \dots \parallel S_n = S$ .
- 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  $W_1 \parallel W_2 \parallel W_3 \parallel W_4 = S_i$
  - 2)  $S_{\text{Reversed}} = W_4 \parallel W_3 \parallel W_2 \parallel W_1$
  - 3)  $T = T \parallel S_{\text{Reversed}}$
- 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:**  $\text{wordbits}$ , the integer 64 or 128  
 $\text{maxwordrange}$ , positive integer less than  $2^{\text{wordbits}}$   
 $\text{key}$ , integer in the range  $0 \dots \text{prime}(\text{wordbits}) - 1$   
 $\text{Msg}$ , string with length divisible by  $(\text{wordbits} / 8)$  octets

**OUTPUT:**  $y$ , integer in the range  $0 \dots \text{prime}(\text{wordbits}) - 1$

- a)  $\text{wordoctets} = \text{wordbits} / 8$
- b)  $p = \text{prime}(\text{wordbits})$
- c)  $\text{offset} = 2^{\text{wordbits}} - p$
- d)  $\text{marker} = p - 1$
- e)  $n = \text{octetlength}(\text{Msg}) / \text{wordoctets}$
- f) Let  $M_1, M_2, \dots, M_n$  be strings of length  $\text{wordoctets}$  octets so that  $\text{Msg} = M_1 \parallel M_2 \parallel \dots \parallel M_n$
- g)  $y = 1$
- h) for  $i = 1$  to  $n$  do
  - 1)  $m = \text{bitstr2uint}(M_i)$
  - 2) if  $(m \geq \text{maxwordrange})$  then
    - i)  $y = (\text{key} * y + \text{marker}) \bmod p$
    - ii)  $y = (\text{key} * y + (m - \text{offset})) \bmod p$
  - 3) else
    - i)  $y = (\text{key} * y + m) \bmod p$
- 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)  $L1Key_i = L1Key [ (i-1) * 16 + 1 \dots (i-1) * 16 + 1024 ]$
  - 2)  $L2Key_i = L2Key [ (i-1) * 24 + 1 \dots i * 24 ]$
  - 3)  $L3Key1_i = L3Key1 [ (i-1) * 64 + 1 \dots i * 64 ]$
  - 4)  $L3Key2_i = L3Key2 [ (i-1) * 4 + 1 \dots i * 4 ]$

- 5)  $A = \text{L1-HASH}(L1Key_i, M)$
  - 6) if  $(\text{bitlength}(M) \leq \text{bitlength}(L1Key_i))$  then
    - i)  $B = 0^{64} \parallel A$
  - 7) else
    - i)  $B = \text{L2-HASH}(L2Key_i, A)$
  - 8)  $C = \text{L3-HASH}(L3Key1_i, L3Key2_i, B)$
  - 9)  $H = H \parallel C$
- 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  $(8 * \text{ceil}(\text{bitlength}(L1Msg)/8192))$  octets

- a)  $t = \max(\text{ceil}(\text{bitlength}(L1Msg)/8192), 1)$
- b) Divide  $L1Msg$  into strings  $M_1, M_2, \dots, M_t$ , so that  $L1Msg = M_1 \parallel M_2 \parallel \dots \parallel M_t$ , and  $\text{octetlength}(M_i) = 1024$  for all  $1 \leq i \leq t-1$ .
- c)  $Len = \text{uint2bitstr}(1024 * 8, 8)$
- d)  $H1 = \langle \text{empty string} \rangle$
- e) for  $i = 1$  to  $t-1$  do
  - 1)  $\text{ENDIAN-SWAP}(M_i)$
  - 2)  $H1 = H1 \parallel (\text{NH}(L1Key, M_i) +_{64} Len)$
- f)  $Len = \text{uint2bitstr}(\text{bitlength}(M_t), 8)$
- g)  $M_t = \text{zeropad}(M_t, 32)$
- h)  $\text{ENDIAN-SWAP}(M_t)$
- i)  $H1 = H1 \parallel (\text{NH}(L1Key, M_t) +_{64} Len)$
- 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)  $Mask64 = \text{uint2bitstr}(0x\ 01FFFFFF\ 01FFFFFF, 8)$

b)  $Mask128 = \text{uint2bitstr}(0x\ 01FFFFFF\ 01FFFFFF\ 01FFFFFF\ 01FFFFFF, 16)$

c)  $k64 = \text{bitstr2uint}(L2Key[1 \dots 8] \wedge Mask64)$

d)  $k128 = \text{bitstr2uint}(L2Key[9 \dots 24] \wedge Mask128)$

e) if  $(\text{octetlength}(L2Msg) \leq 2^{17})$  then

1)  $y = \text{POLY}(64, 2^{64} - 2^{32}, k64, L2Msg)$

f) else

1)  $M_1 = L2Msg[1 \dots 2^{17}]$

2)  $M_2 = L2Msg[2^{17} + 1 \dots \text{octetlength}(L2Msg)]$

3)  $M_2 = \text{zeropad}(M_2 \parallel \text{uint2bitstr}(0x80, 1), 16)$

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

5)  $y = \text{POLY}(128, 2^{128} - 2^{96}, k128, \text{uint2bitstr}(y, 16) \parallel M_2)$

g)  $H2 = \text{uint2bitstr}(y, 16)$

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)  $M_i = Msg [(i - 1) * 2 + 1 \dots i * 2]$

ii)  $K_i = K1[(i - 1) * 8 + 1 \dots i * 8]$

iii)  $m_i = \text{bitstr2uint}(M_i)$

iv)  $k_i = \text{bitstr2uint}(K_i) \bmod \text{prime}(36)$

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

1)  $y = (m_1 * k_1 + \dots + m_8 * k_8) \bmod \text{prime}(36)$

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

3)  $H3 = \text{uint2bitstr}(y, 4)$

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<sup>[10]</sup>.

### 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.<sup>[3]</sup>. It is based on the NH function of Black et al.<sup>[2]</sup> 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)  $k_L = \text{octetstr2uint}(LKey[0..3])$ ,  $k_U = \text{octetstr2uint}(LKey[4..7])$
- b)  $m1_L = \text{octetstr2uint}(Right[0..3])$ ,  $m1_U = \text{octetstr2uint}(Right[4..7])$
- c)  $m2 = \text{octetstr2uint}(Left[0..7])$
- 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)  $LHash = \text{uint2octetstr}(h', 8)$
- 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:

- $\text{PRG\_Init}(K, N)$  initializes the inner state of the PRG with key  $K$  and nonce  $N$ .
- $\text{PRG\_Next}(n)$  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)  $u = \text{taglen} / 4$
- d)  $v = \max\{1, \text{ceil}(\log_2(\text{maxlen})) - 6\}$
- e) for  $j = 1$  to 6:
  - 1) for  $i = 1$  to  $u$ :
    - i)  $kf_{j,i} = \text{octetstr2uint}(\text{PRG\_Next}(32))$
    - ii) words\_used = words\_used + 1
- f) for  $j = 1$  to 6:
  - 1) for  $i = 1$  to  $u$ :
    - i) while( $k_{j,i} \geq \text{prime}(32)$ )
      - I)  $kf_{j,i} = \text{octetstr2uint}(\text{PRG\_Next}(32))$
      - II) words\_used = words\_used + 1
- g) while(words\_used mod 4  $\neq$  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)  $KL_{j,i} = \text{PRG\_Next}(64)$
- i) PRG\_Init( $K$ ,  $N$ )
- j)  $K_E = \text{PRG\_Next}(32 \cdot u)$
- 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  $\text{taglen}$ , integer (one of 4,8,12,16,20)

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

- a)  $len = \text{bitlength}(M)$  as 64-bit integer
- b) if  $len = 0$ :
  - 1)  $M_1 = \dots = M_u = 0^{64}$
- 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)  $v' = \max\{1, \text{ceil}(\log_2(len)) - 6\}$
    - iii) for  $j = 1$  to  $v'$ :
      - I)  $t = \text{octetlength}(M_i) / 8$
      - II) Divide  $M_i$  into 8-octet blocks  $B_1, \dots, B_t$  such that  $M_i = B_t \parallel \dots \parallel B_1$
      - III) if  $t$  is even:
        - a)  $M_i = \text{ENH}(KL_{j,i}, B_t, B_{t-1}) \parallel \dots \parallel \text{ENH}(KL_{j,i}, B_2, B_1)$
      - IV) else:
        - a)  $M_i = B_t \parallel \text{ENH}(KL_{j,i}, B_{t-1}, B_{t-2}) \parallel \dots \parallel \text{ENH}(KL_{j,i}, B_2, B_1)$
  - d) for  $i = 1$  to  $u$ :
    - 1)  $Q_i = 0^7 \parallel len \parallel M_i$
    - 2) Divide  $Q_i$  into 27-bit blocks  $B_1, \dots, B_5$  such that  $Q_i = B_5 \parallel \dots \parallel B_1$ .
    - 3) Pad each block  $B_1, \dots, B_5$  with zeroes in the most significant bits such that it is 4 octets long.
    - 4) for  $j = 1$  to 5:
      - i)  $b_j = \text{octetstr2uint}(B_j)$
    - 5)  $s_i = (b_1 * kf_{1,i}) + \dots + (b_5 * kf_{5,i}) + kf_{6,i} \bmod \text{prime}(32)$
    - 6)  $S_i = \text{uint2octetstr}(s_i, 4)$
  - e)  $H = S_u \parallel \dots \parallel S_1$
  - 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)  $K_H = K[0 \dots 15]$

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

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  $c_1, \dots, c_s$ , sequence of 17-octet integers

- a) Let  $l_o = \text{octetlength}(M)$
- b) Let  $s = \text{ceil}(l_o/16)$
- c) Let  $t = \text{floor}(l_o/16)$
- d) For  $i = 0, \dots, t-1$ :
  - 1)  $c_{i+1} = \text{octetstr2uint}(M[16i \dots 16i+15]) + 2^{128}$
- e) If  $s > t$ :
  - 1) Let  $r = l_o \bmod 16$
  - 2)  $c_s = \text{octetstr2uint}(M[16t \dots l_o-1]) + 2^{8r}$
- f) Output  $s; c_1, \dots, c_s$

#### 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  $c_1, \dots, c_s$ , sequence of 17-octet integers

**OUTPUT:** Hash value  $H$ , 16-octet integer

- a)  $r = \text{octetstr2uint}(K_H)$
- b)  $H' = (c_1 * r^s + c_2 * r^{s-1} + \dots + c_s * r^1) \bmod \text{prime}(130)$
- c) Let  $H = H' \bmod 2^{128}$
- 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  $S = \text{AES-128}(K_E, N)$
- b)  $s = \text{octetstr2uint}(S)$
- c) Let  $mac = (H + s) \bmod 2^{128}$
- d)  $MAC = \text{uint2octetstr}(mac, 16)$
- e) Output  $MAC$

## 6.5 GMAC

### 6.5.1 Description of GMAC

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 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  $|K|$ .
- A tag length  $t$  in bits, where  $64 \leq t \leq 128$ .
- 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  $\text{bitlength}(W) = 128(k-1)+u$  and  $0 < u \leq 128$ . Let  $W_1, W_2, \dots, W_k$  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  $\text{bitlength}(Z) = 128(l-1)+v$  and  $0 < v \leq 128$ . Let  $Z_1, Z_2, \dots, Z_l$  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)  $X_i = (X_{i-1} \oplus W_i) \bullet H$ ,  $1 \leq i \leq k-1$  (this step is omitted if  $k \leq 1$ ).
  - 3)  $X_k = (X_{k-1} \oplus (W_k \parallel 0^{128-u})) \bullet H$  (this step is omitted if  $k=0$ ).
  - 4)  $X_i = (X_{i-1} \oplus Z_{i-k}) \bullet H$ ,  $k+1 \leq i \leq k+l-1$  (this step is omitted if  $l \leq 1$ ).
  - 5)  $X_{k+l} = (X_{k+l-1} \oplus (Z_l \parallel 0^{128-v})) \bullet H$  (this step is omitted if  $l=0$ ).
  - 6)  $X_{k+l+1} = (X_{k+l} \oplus \text{uint2bitstr}(\text{bitlength}(W),8) \parallel \text{uint2bitstr}(\text{bitlength}(Z),8)) \bullet H$ .
- 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)  $H = \text{GHASH}(K_H, M, \{\})$
- 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  $\text{bitlength}(N) = 96$  then let  $Y_0 = N \parallel 0^{31} \parallel 1$ . Otherwise let  $Y_0 = \text{GHASH}(K_H, \{\}, N)$ .
- b) Let  $\text{MAC} = (H \oplus \text{Enc}(K_E, Y_0))|_t$ .
- c) Output MAC.